Oracle DBA实践操作指南

Oracle PL/SQL DBA编程入门

Learn Oracle from Oracle Certified Master

林树泽 编著



完整、详尽、真实的Oracle数据库实用指导

- 全面讲解PL/SQL语言编程的基础知识和技巧
- 通过示例说明概念和基本操作，方便读者把握相应的内容
- 使用Oracle数据库默认安装的SAMPLE表，减少学习难度和操作复杂性
- 结合笔者Oracle数据库编程和维护经验，使读者入门更容易
- 指导读者学会编写自己的数据库监控工具
- 学习编程的最终方法是实践，运行实例有助于深入理解所学内容

清华大学出版社

Oracle
PL/SQL DBA编程入门

林树泽　编著



清华大学出版社
北　京

内 容 简 介

本书从 Oracle 的数据库体系结构出发，全面讲解 PL/SQL 编程的运行原理、基本概念和编程技巧，是学习 Oracle PL/SQL 编程的入门教材。

本书内容包括 17 章，全面讲解了 PL/SQL 语言的编程环境、数据类型、流程控制、游标、触发器、存储过程、函数、包、异常、记录、集合类型以及 Oracle 常用工具包等内容。

本书适合 PL/SQL 编程的初学者，包括 Oracle 数据库应用开发人员、Oracle 数据库设计人员、Oracle DBA 等阅读，也适合高等院校和培训学校相关专业的师生作为教学参考。

图书在版编目（CIP）数据

Oracle PL/SQL DBA 编程入门/林树泽编著.–北京：清华大学出版社，2013
ISBN 978-7-302-33382-1

Ⅰ. ①O…　Ⅱ. ①林…　Ⅲ. ①关系数据库系统－程序设计　Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字（2013）第 180856 号

责任编辑：夏非彼
封面设计：王　翔
责任校对：闫秀华
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**19.25　　**字　　数：**493 千字
版　　次：2013 年 10 月第 1 版　　**印　　次：**2013 年 10 月第 1 次印刷
印　　数：1~4000
定　　价：45.00 元

产品编号：053930-01

前　言

关于本书

本书是学习 Oracle PL/SQL 编程的入门教材，面向的对象是 Oracle 数据编程的初学者，或因正在学习 Oracle 数据库管理而需要学习 PL/SQL 编程相关知识的初级读者。本书在讲解方式上强调基础，即基本概念、基本原理和基本操作，与其他 PL/SQL 编程书籍不同，本书介绍了 Oracle 数据库的体系结构，对架构的初步了解有助于读者理解 PL/SQL 程序的运行原理，这样读者不仅理解了 PL/SQL 编程的基本方法，同时也理解了 PL/SQL 程序的运行载体。读者可以根据需要独立学习每章的基本概念和基本编程操作。

考虑到初学者的需求，本书提供了大量的示例程序、运行结果，同时对示例都有详细的注释，所以只要读者理解了基础概念的内容和基本操作的方法，再阅读代码并亲自运行示例就很容易掌握 PL/SQL 编程的知识。其实，要求读者掌握的内容已在示例程序中体现出来，所以在示例程序的选用和注释方面笔者充分考虑了初学者的特点。

本书介绍的知识领域比较全面。讲解了 PL/SQL 语言的使用环境、简要发展历史、数据类型、语言流程控制、异常处理以及 PL/SQL 调试等，这些对于编写基本的 PL/SQL 程序逻辑很重要，同时也包含了触发器、存储过程、函数、包以及 Oracle 常用工具包等，所以 PL/SQL 编程人员以及数据库管理人员都可以从书中获得相应的基础知识。

本书的特点

在撰写本书的前言之前，笔者又浏览了一遍书稿。就知识点而言本书还是比较全面的，示例较多，通过示例说明概念和基本操作是笔者坚持的方法，学习编程最终是实践，理解理论最佳的方式也是实践，即根据业务逻辑编写代码，从而了解 PL/SQL 编程的基础知识和基本概念，并且示例都有详细的注释，读者只要阅读该示例，再尝试运行即可掌握相应的内容。

本书基本涵盖了 Oracle 数据库 PL/SQL 编程的全部基础知识。为了方便初学者的学习，笔者在本书开始处便介绍了如何快速安装 Oracle 数据库，这样读者就拥有了一个学习和练习的环境，利用 Oracle 数据库自带的 Sample 示例便可完成书中所有的练习。

本书的特点主要体现在以下几个方面。

- 本书的编排采用循序渐进的方式，示例程序丰富，注释清晰，适合初、中级读者逐步掌握 Oracle 数据库 PL/SQL 编程的基础知识，以提高读者使用 PL/SQL 语言在实际工作中的能力。
- 本书在介绍示例时，采用了浅显易懂的实例，并且都使用 Oracle 数据库默认安装的 SAMPLE 表来实现各种 SQL 操作和相关概念的讲解，这样读者就不必再创建大量的表，从而减少读者的学习难度和操作复杂性，对于初学者尤为适用。

- 本书在着重介绍 Oracle 数据库 PL/SQL 编程的各方面知识外，还加入了数据库工具的内容，这部分内容使得读者能够学会编写自己的数据库监控工具，当然若想掌握这些监控和数据库维护工具的编写还需要了解数据库架构以及对应的知识点，但是书中主要介绍了这些监控工具编写的基本方法和基本思路，希望能够在读者的实际工作中发挥辅助作用。
- 本书结合笔者在 Oracle 数据库编程和数据库维护方面的经验，在各个章节的介绍中都从初学者的角度出发进行讲解，充分考虑了初学者的特点，使得读者入门更容易，轻松上手学习 Oracle 数据库 PL/SQL 编程。

学习方法和读者对象

学习 Oracle 的 PL/SQL 编程自然不是一件轻松的事情，其实学习任何知识都需要读者自己的付出。但是作为一本 PL/SQL 编程的参考书，希望读者在理解基本概念后，务必要将书中的实例“跑”一遍，对照代码以及计算结果，再辅以实例解释，从而轻松理解书中的基本概念以及工具的使用方法。

本书的目标读者是 PL/SQL 编程的初学者，包括 Oracle 数据库应用开发人员、Oracle 数据库设计人员、Oracle DBA 等。希望读者可以多花一些时间学习书中介绍的每一章的基础知识，这些是 PL/SQL 编程的基础，基础不牢地动山摇，扎实的基础是我们继续前进的基石。

除了封面署名作者外，参与本书编写的人员还有历铁帅、何会军、李渊、陈玉等，为本书的创作，他们也做了大量的工作，在此表示衷心的感谢。

编　者

2013 年 8 月

目　录

第 1 章

PL/SQL编程环境

在本书的开始部分，我们先介绍一下 PL/SQL 的编程环境，并创建一个数据库，使用这个数据库实例的 PL/SQL 接口，即 SQL*Plus 来完成编程工作，同时在 HR 模式下完成本书所有的实例。本章只需要读者熟悉这个环境，然后理解如何连接数据库、如何连接到特定的数据库模式并执行 DML 以及 DDL 操作即可。这些都是我们后续学习的基础和平台。

1.1 创建数据库

首先创建一个测试数据库，数据库实例名为 ORCL，如图 1-1 所示。下面是在创建数据库时的一些关键步骤。



图 1-1　设置数据库全局服务名和 SID。

在上图中，我们设置 Oracle 数据库的全局服务名为 ORCL，SID 为 ORCL。选择在创建数据库时安装 Enterprise Manager（企业管理器），如图 1-2 所示。

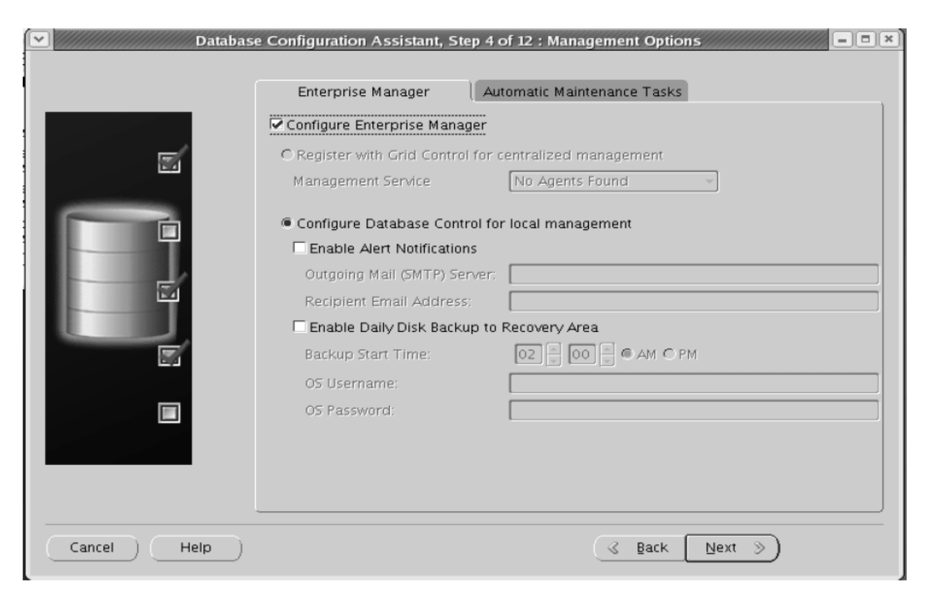


图 1-2　安装 Enterprise Manager

单击上图中的 Next 按钮，在新的对话框中设置用户密码，如图 1-3 所示。



图 1-3　设置用户密码

此时，系统提示设置用户密码，这里我们对所有账户使用了相同的用户密码，此时的密码为 oracle，至于其他模式的密码，我们可以使用 SYS 权限进行修改。

为了在我们创建的数据库中已有现成的用户以及数据可用，例如 HR 用户，以及它拥有的表，在这里我们选中 Sample Schemas 复选框，如图 1-4 所示。这样设置后在学习 PL/SQL 编程时就不需要额外创建其他表了（但不是绝对的）。

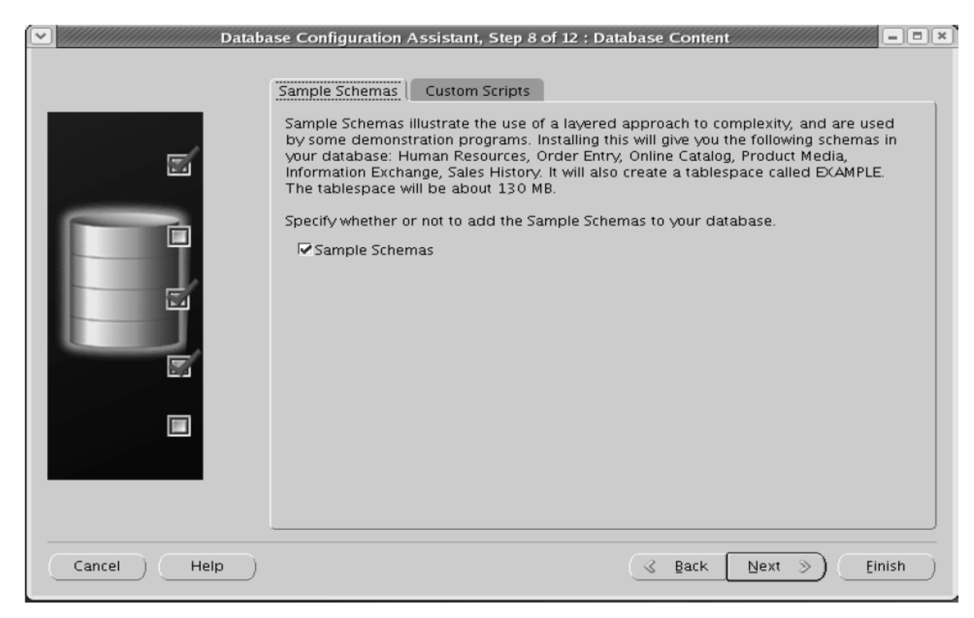


图 1-4　选中示例模式

在图 1-5 中，我们选择当前数据库的内存分配，即 Oracle 占用的系统内存的比例，我们选择默认设置，字符集和连接模式也选择默认方式，字符集的默认值为操作系统的字符集，而连接模式的默认方式为专有连接，如图 1-6 和图 1-7 所示。

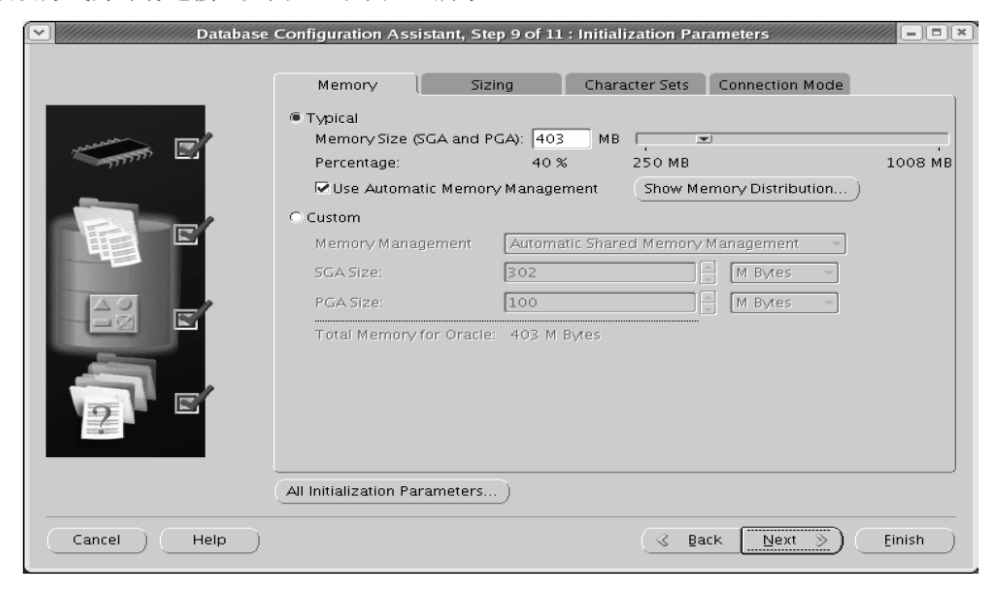


图 1-5　选择内存大小

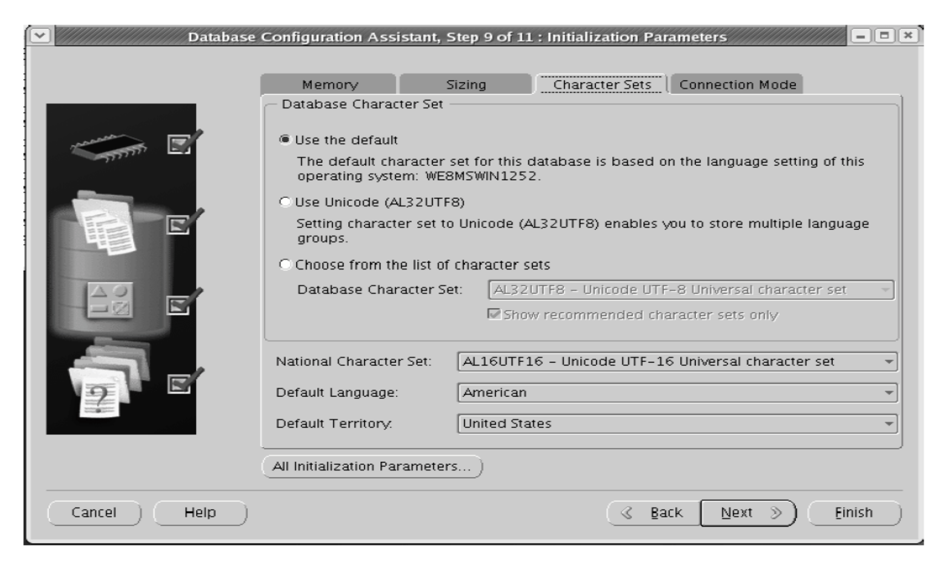


图 1-6　选择数据库字符集

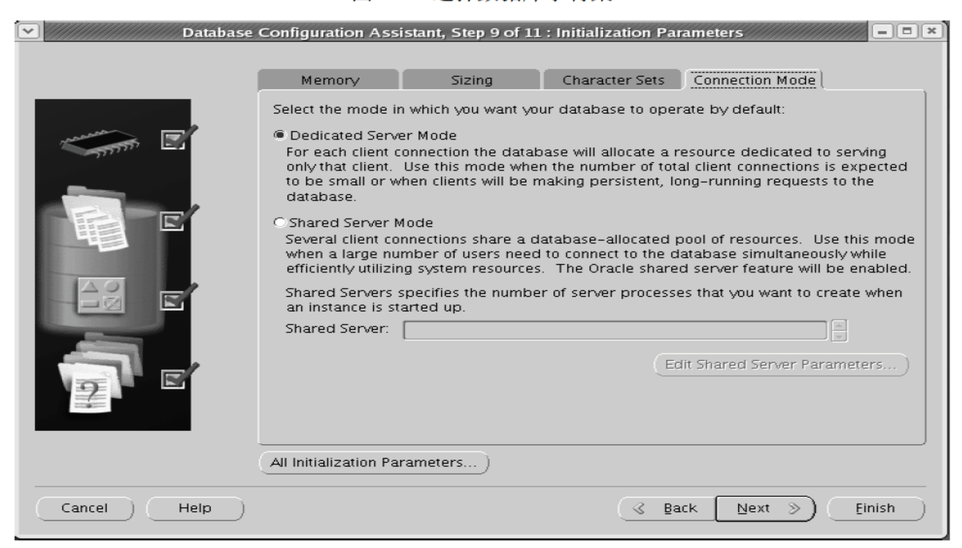


图 1-7　选择数据库连接模式

以上是创建数据库时的关键步骤，其余步骤只要按照提示进行顺序操作即可，这样我们就完成了数据库 ORCL 的创建，为我们学习 PL/SQL 编程创建了基础平台。

1.2　简述实例、服务器与物理结构

Oracle 数据库中有几个概念必须清楚，即 Oracle 服务器、Oracle 实例以及 Oracle 物理结构。

Oracle 服务器和实例是非常重要的两个概念，这里的服务器不仅仅是一个物理概念，还包括系统进程，而实例则是 DBA 经常维护的对象。

1.2.1　Oracle 实例

Oracle 实例由一些内存区和后台进程组成，如图 1-8 所示，内存区包括数据库高速缓存、重做日志缓存、共享池、流池以及其他可选内存区（如 Java 池），这些池也称为数据库的内存结构；后台进程包括系统监控进程（SMON）、进程监控（PMON）、检验点进程（CKPT）、数据库写进程（DBWR）、日志写进程（LGWR）、归档进程（ARCn）、其他进程（OTHERS）等，这些数据库系统进程忠于职守、相互协作，从而完成数据管理任务。

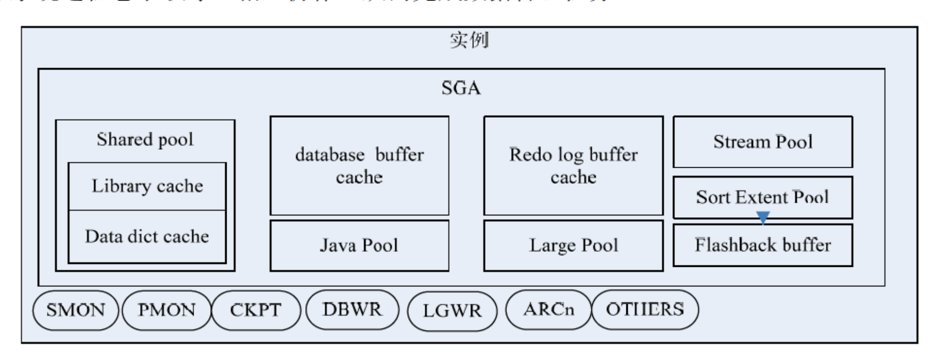


图 1-8　Oracle 实例（Instance）组成图

若要访问数据库必须先启动实例，启动实例时，先分配内存区，然后再启动后台进程，后台进程执行数据库的输入、输出操作以及监控其他 Oracle 进程。在数据库启动过程中有 5 个进程是必须启动的，它们是系统监控进程（SMON）、进程监控（PMON）、数据库写进程（DBWR）、日志写进程（LGWR）、检验点进程（CKPT），否则实例无法创建。在数据库启动过程中我们可以在告警日志（alertSID.ora）中看到详细的过程。



在实践过程中，为了方便操作，会通过数据库工具在计算机重启时自动启动数据库，如果用户安装了其他占用大量内存的应用软件，可能会造成数据库启动失败，此时往往是因为内存不足，操作系统无法为 Oracle 分配 SGA 所致，这时必须的 5 个进程也无法启动。

下面我们通过操作系统工具查询一下当前的实例进程，如实例 1-1 所示。

实例 1-1　查询实例进程。

```
[oracle@localhost ~]$ ps -ef |grep _ORCL
oracle    7890     1  0 18:16 ?        00:00:00 ora_pmon_ORCL
oracle    7892     1  0 18:16 ?        00:00:00 ora_vktm_ORCL
oracle    7896     1  0 18:16 ?        00:00:00 ora_gen0_ORCL
oracle    7898     1  0 18:16 ?        00:00:01 ora_diag_ORCL
oracle    7900     1  0 18:16 ?        00:00:00 ora_dbrm_ORCL
oracle    7902     1  0 18:16 ?        00:00:00 ora_psp0_ORCL
oracle    7904     1  0 18:16 ?        00:00:00 ora_dia0_ORCL
oracle    7906     1  2 18:16 ?        00:00:05 ora_mman_ORCL
oracle    7908     1  0 18:16 ?        00:00:00 ora_dbwr_ORCL
oracle    7910     1  0 18:16 ?        00:00:01 ora_lgwr_ORCL
oracle    7912     1  0 18:16 ?        00:00:00 ora_ckpt_ORCL
oracle    7914     1  0 18:16 ?        00:00:01 ora_smon_ORCL
oracle    7916     1  0 18:16 ?        00:00:00 ora_reco_ORCL
```

```
oracle    7918     1  0 18:16 ?        00:00:02 ora_mmon_ORCL
oracle    7920     1  0 18:16 ?        00:00:00 ora_mmnl_ORCL
…………
oracle    8199  8057  0 18:20 pts/2     00:00:00 grep _ORCL
```

在上例中包含了实例要求的必须启动的 5 个进程，当然这里显示出实例中还有其他进程，这些进程分别用于完成其他任务，如诊断性进程 DIAG 等。

1.2.2 Oracle 服务器

Oracle 服务器由数据库实例和数据库文件组成，也就是我们经常说的数据库管理系统（DBMS）。数据库服务器的组成如图 1-9 所示。

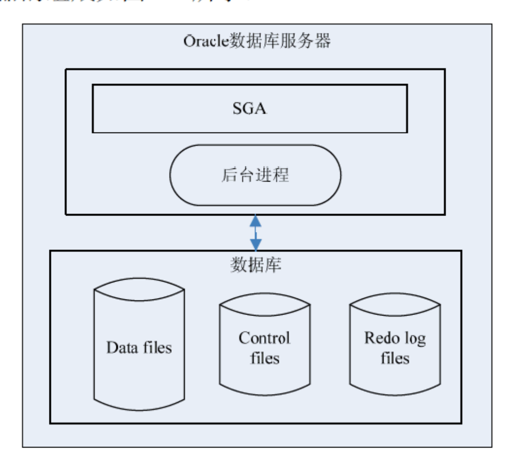


图 1-9 Oracle 服务器组成

数据库服务器用于完成对数据的操作，任何对数据库的访问都必须通过实例来完成，实例的组件再访问数据库文件或直接在内存中操作。数据库服务器除了维护实例和数据库文件外，还在用户建立与服务器的连接时启动服务器进程并分配 PGA。

1.2.3 Oracle 物理结构

我们都知道，数据库是运行在操作系统之上的，数据库的最终目的就是存储和获取相关的数据，这些数据实际上是存储在操作系统文件中，这些操作系统文件组成了 Oracle 数据库的物理结构。

Oracle 数据库的物理结构是指数据库中的一系列操作系统文件，Oracle 数据库由三类文件组成。

- 数据文件（Data File）：数据文件包含数据库中的实际数据，是数据库操作中数据的最终存储位置。
- 控制文件（Control File）：包含维护数据库和验证数据库完整性的信息，它是二进制文件。
- 重做日志文件（Redo File）：重做日志文件包含数据库发生变化的记录，在发生故障时用于数据恢复。

当我们连接数据库时，实际上是连接到数据库实例，实例的某些进程再具体完成操作数据库(这

里所说的数据库就是指数据文件）以及与用户交互的任务。

创建的 PL/SQL 程序最终的操作对象就是数据文件，因为只有数据文件才能存放用户数据。例如实例 1-2 用于查询当前数据库的数据文件。

实例 1-2　查询数据文件。

```
SQL> select file_id,file_name,tablespace_name from dba_data_files
  2  order by 1;

   FILE_ID  FILE_NAME                                          TABLESPACE_NAME
---------- -------------------------------------------------- -----------
         1  /u01/app/oracle/oradata/ORCL/system01.dbf           SYSTEM
         2  /u01/app/oracle/oradata/ORCL/sysaux01.dbf           SYSAUX
         3  /u01/app/oracle/oradata/ORCL/undotbs01.dbf          UNDOTBS1
         4  /u01/app/oracle/oradata/ORCL/users01.dbf            USERS
         5  /u01/app/oracle/oradata/ORCL/example01.dbf          EXAMPLE
```

上面的数据文件是在创建数据库时默认需要创建的数据文件，其中文件 1 和文件 2 是必须的，文件 5 用于存放示例模式中的数据，如 HR 模式的数据就存放在这个表空间。我们使用 HR 模式登录数据，查询经常使用到的 EMPLOYEES 表，看看该表所在的表空间是否为 EXAMPLE，如实例 1-3 所示。

实例 1-3　查询表 EMPLOYEES 所在表空间。

```
SQL> connect hr/oracle
Connected.
SQL> select tablespace_name from user_tables where table_name='EMPLOYEES';

TABLESPACE_NAME
---------------
EXAMPLE
```

1.3　认识连接与会话

连接与会话是 Oralce 数据库中容易混淆的两个概念，它们是紧密相关的。下面讲解一下它们的区别，并给出实际的实例以帮助读者更好地理解它们的概念。

1.3.1　连接

连接是指用户进程与数据库服务器之间的通信途径，一个连接可以有多个对话。Oracle 提供了三种数据库连接方式，以满足用户不同的连接需求，三种连接方式如下。

- 基于主机的方式（Host-Based）：在此方式中，服务器和客户端运行在同一台计算机上，用户可以直接连接数据库服务器。
- 基于客户机-服务器的方式（Client-Server）：在此方式中，数据库服务器和客户端运行在不同的计算机上，客户通过网络连接数据库服务器。在 DBA 的日常维护中，会经常使用这种方式访问数据库，以实现数据库的远程维护。
- 用户-应用服务器-数据库服务器方式（Client-Application Server-Server）:这种方式称为三

层访问模式，用户首先访问应用服务器，然后由应用服务器连接数据库服务器，应用服务器就像一个中介，用于完成客户和数据库的交互。在很多应用系统中，客户的应用程序往往通过三层方式访问数据库，如应用服务器为 IIS 或 Apache 服务器等。

1.3.2 会话

会话是指一个明确的数据库连接。在用户与数据库服务器建立连接后，一旦用户采用一种连接方式，我们就把这样的连接称为一个会话。

如果用户通过某种工具（如 SQL*Plus）在专有连接的情况下访问数据库，在输入的用户名和密码经过服务器验证后，服务器就会自动创建一个与该用户进程对应的服务器进程，二者是一对一的关系，这里的服务器进程就像用户进程的代理，代替用户进程向数据库服务器发出各种请求，并把从数据库服务器获得的数据返回给用户进程。

在用户退出或发生异常时（操作系统重启）会话结束。



专有连接是一种连接类型，是指用户和服务器进程之间一对一的关系。而在共享服务器配置的情况下，多个用户进程可以同时共享服务器进程，此时就不是专有连接，而是多对一的关系。

一个用户可以并发地建立多个会话，实例 1-4 就是用户 SYS 同时在专有连接的情况下建立两个会话的实例。

实例 1-4　用户 SYS 通过专有连接建立两个会话。

```
SQL>SELECT serial#,username,status,server,process,program,logon_time
  2* FROM v$session
SERIAL# USERNAME STATUS  SERVER    PROCESS   PROGRAM   LOGON_TIME
------- ------------- ----- ------------ --------- -------- --------
      1          ACTIVE  DEDICATED 2172      ORACLE.EXE       17-5 月-11
      1          ACTIVE  DEDICATED 528       ORACLE.EXE       17-5 月-11
      1          ACTIVE  DEDICATED 2188      ORACLE.EXE       17-5 月-11
      1          ACTIVE  DEDICATED 408       ORACLE.EXE       17-5 月-11
      1          ACTIVE  DEDICATED 1424      ORACLE.EXE       17-5 月-11
      1          ACTIVE  DEDICATED 1244      ORACLE.EXE       17-5 月-11
      1          ACTIVE  DEDICATED 2264      ORACLE.EXE       17-5 月-11
      3 SYS      ACTIVE  DEDICATED 540:1644   sqlplus.exe     17-5 月-11
      7 SYS      ACTIVE  DEDICATED 1296:1848  sqlplusw.exe    17-5 月-11

已选择 9 行。
```

在实例 1-4 的输出中可以看到，我们同时使用两个 SQL*Plus 工具连接数据库，并且使用同一个用户 SYS，最后两行显示有两个活跃（ACTIVE）的会话：一个会话使用 sqlplus.exe 程序建立；另一个使用 sqlplusw.exe 程序建立。



在上述查询中，只是演示一个用户可以建立多个连接，使用相同或不同的工具登录。至于 v$session（数据字典视图），读者可暂且把它看成是一张表，表中存储了当前会话的信息，如属性 USERNAME 是用户登录名，属性 PROGRAM 是用户登录工具（一个用户进程）。

图 1-10 清晰地说明了连接与会话之间的区别和联系。

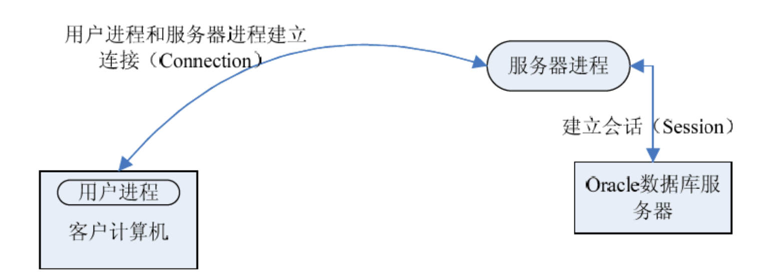


图 1-10　连接与会话示意图



一个连接可以对应多个会话，连接仅仅是一种通信途径，如通过 Socket 建立通信，但是一个用户可以启动多个进程，这里的服务器进程就像是用户进程的代理一般，与服务器交互完成数据的各种操作。

1.3.3　建立到数据库的连接

我们在学习 PL/SQL 编程时，最重要的工具就是使用 Oracle 的 SQL*Plus，这是一个客户端工具，作为用户和数据库之间的接口来操作数据库。SQL*Plus 是集成在 Oracle 数据库软件中的工具，下面我们演示如何通过 SQL*Plus 来连接数据库。

首先启动 SQL*Plus 工具，SQL*Plus 工具是 Oracle 提供的一个用户接口，可通过该接口来完成用户与数据库之间的数据操作，代码如实例 1-5 所示。

实例 1-5　启动 SQL*Plus 工具。

```
[oracle@localhost ~]$ sql*plus /nolog

SQL*Plus: Release 11.2.0.1.0 Production on Sat Dec 3 18:40:14 2011

Copyright (c) 1982, 2009, Oracle.  All rights reserved.

SQL>
```

此时打开 SQL*Plus 工具，前面已经说过 SQL*Plus 是 Oracle 的一个工具，位于 $ORACLE_HOME/bin 目录下。在.bash_profile 文件中使用 export 指令即可指明路径，所以我们输入 sqlplus 指令，当前操作系统会自动搜索该指令，.bash_profile 文件的内容如实例 1-6 所示。

实例 1-6　.bash_profile 文件的内容。

```
[oracle@localhost ~]$ cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

```
# User specific environment and startup programs
export ORACLE_BASE=/u01/app/oracle/
export ORACLE_HOME=/u01/app/oracle/product/11.2.0/dbhome_1
export ORACLE_SID=ORCL
PATH=$PATH:$HOME/bin
export PATH=$ORACLE_HOME/bin:$PATH
unset USERNAME
```

以上内容已经解释了 sqlplus 工具。下面我们连接到数据库，此时需要一个有效的用户名和合法的密码，例如连接到 SYS 用户，该用户必须指定为 sysdba 角色，代码如实例 1-7 所示。

实例 1-7　连接到 SYS 用户。

```
SQL> connect sys/oracle as sysdba
Connected.
```

此时，我们已经连接到数据库，可以执行任何需要的数据库操作，当然此时也可以编写 PL/SQL 程序。如果当前有多个数据库实例，还可以通过如下方式连接数据库，需要先指定要连接的数据库 ID，代码如实例 1-8 所示。

实例 1-8　连接数据库。

```
[oracle@localhost ~]$ export ORACLE_SID=ORCL
[oracle@localhost ~]$ sqlplus sys/oracle as sysdba

SQL*Plus: Release 11.2.0.1.0 Production on Sat Dec 3 18:46:49 2011

Copyright (c) 1982, 2009, Oracle.  All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

1.4　简述HR模式

在安装数据库时已安装了 SAMPLES，此时会创建一系列模式，以便用户学习和试验，例如可通过实例 1-9 查询 HR 模式的用户状态。

实例 1-9　查询 HR 模式的用户状态。

```
SQL> select username,account_status from dba_users;

USERNAME                          ACCOUNT_STATUS
------------------------------ --------------------------------
SYS                              OPEN
SYSTEM                           OPEN
SCOTT                            OPEN
HR                               EXPIRED & LOCKED
……
36 rows selected.
```

显示用户状态为 EXPIRED & LOCKED，即需要解锁该用户，解锁 HR 用户的代码如实例 1-10

所示。

实例 1-10　解锁用户 HR。

```
SQL> alter user hr account unlock identified by oracle;

User altered.
```

此时已解锁用户 HR，并且设置用户密码为 oracle。图 1-11 是 HR 模式中表的关系图。

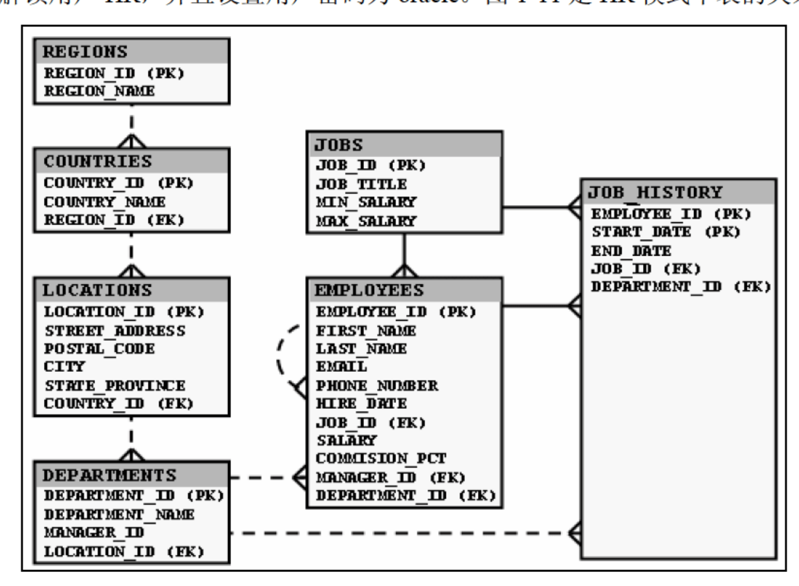


图 1-11　HR 模式中表的关系图

本节使用的示例来自一个人力资源（HR）应用程序，可以将此应用程序创建为启动数据库的一部分。下面是该 HR 应用程序的主要业务规则：

- 每个部门可以雇佣一个或多个雇员。每个雇员被分配到一个（且仅一个）部门。
- 每个职务必须是一个或多个雇员的职务。当前必须已为每个雇员分配了一个（且仅一个）职务。
- 当一个雇员更改了其部门或职务时，JOB_HISTORY 表中的某一条记录会记录以前分配的开始日期和结束日期。
- JOB_HISTORY 记录由组合主键（PK），即 EMPLOYEE_ID 和 START_DATE 列标识。

实线表示必须使用的外键（FK）约束条件，虚线表示可选的 FK 约束条件。EMPLOYEES 表自身也有一个 FK 约束条件。下面是一个实施业务规则：每个雇员可以直接向一个（且仅一个）经理报告工作。FK 是可选项，原因是最高职位的雇员不用向其他雇员报告工作。

我们可以通过如实例 1-11 所示的代码查询当前用户拥有的所有表。

实例 1-11　查询当前用户拥有的表。

```
SQL> select * from cat where table_type='TABLE';

TABLE_NAME                     TABLE_TYPE
```

```
------------------------------ -----------
REGIONS                        TABLE
LOCATIONS                      TABLE
JOB_HISTORY                    TABLE
JOBS                           TABLE
EMPLOYEES                      TABLE
DEPARTMENTS                    TABLE
COUNTRIES                      TABLE

7 rows selected.
```

利用同义词 CAT 可查询当前数据库的表对象，输出的表名和图 1-11 中的一致，表之间的关系可以通过查询外键关系确立，感兴趣的读者可以自行确认。

1.5　本章小结

本章介绍了学习 PL/SQL 编程的配置环境，首先创建了一个 Oracle 11g 版本的数据库，并给出了涉及的关键步骤，然后介绍了数据库实例、数据库服务器以及数据库物理结构的概念，通过数据字典或操作系统工具可方便地查询这些信息。连接和会话也是十分重要的概念，二者相互关联，通过本章的介绍，读者可以知道如何使用 SQL*Plus 工具连接到数据库。最后介绍了 HR 模式的信息，书中介绍的 PL/SQL 编程实例都是基于该模式的表。

第 2 章

◂ PL/SQL基本概念 ▸

PL/SQL 语言是对 SQL 语言的功能扩充，SQL 语言适合管理关系型数据库，但是无法满足应用程序对数据更复杂的处理需求。PL/SQL 语言用于创建存储过程、函数、触发器、PL/SQL 包以及用户自定义函数。Oracle PL/SQL 在企业级应用程序中应用广泛，而且 Oracle 的一些功能部件也是使用 PL/SQL 编写的。

2.1 PL/SQL的应用环境

PL/SQL 适用于客户端与服务器端的开发，具有高级语言所拥有的编程结构，使用 PL/SQL 可极大地增加数据库编程的灵活性。PL/SQL 不是独立的编程语言，它是 Oracle 数据库服务器的一部分，可以在服务器和客户端两种环境中运行。无论在哪种环境中运行都需要依赖 PL/SQL 引擎进行处理。

PL/SQL 引擎在服务器端的 PL/SQL 语句处理过程如图 2-1 所示。

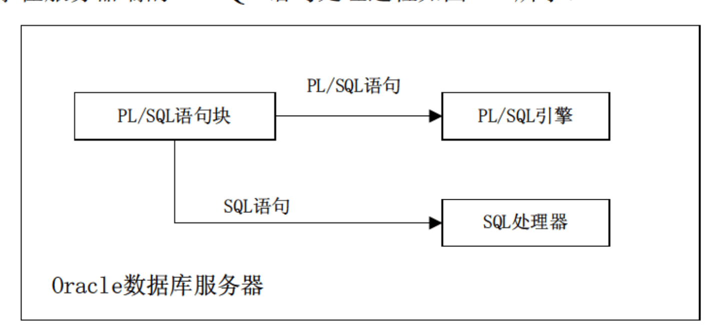


图 2-1 处理过程

SQL 语句与 PL/SQL 语句的处理是不同的，均由各自的引擎处理，当 PL/SQL 语句在数据库服务器端执行时，PL/SQL 引擎会将 PL/SQL 语句与 SQL 语句分开，PL/SQL 语句块会由 PL/SQL 引擎处理，而 SQL 语句就由 PL/SQL 引擎送到 SQL 语句处理器进行处理，SQL 处理器一般处于数据库服务器端。

PL/SQL 数据库引擎在客户端的 PL/SQL 语句处理过程如图 2-2 所示。

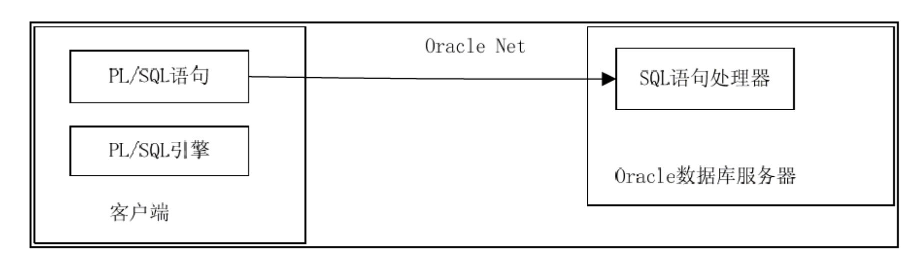


图 2-2　语句处理过程

当 PL/SQL 引擎处于客户端时，此时所有的 PL/SQL 语句均在客户端执行，而 SQL 语句会送到数据库服务器进行处理，SQL 语句处理器位于服务器。如果 PL/SQL 语句不包含 SQL 语句，则整个 PL/SQL 程序均在客户端执行。

在传统的客户端-服务器应用程序中，如果没有 PL/SQL 程序，那么在执行 SQL 语句时就会出现一些问题。因为 PL/SQL 程序块会将 SQL 语句集中在一起执行，这样在客户端就可以将整个需要执行的 SQL 语句传输到服务器，然后将所有 SQL 语句的执行结果再返回到客户端，如果没有使用 PL/SQL 程序，则需要多次传输 SQL 语句，每次执行后再返回数据的方式显然增加了网络流量。而使用 PL/SQL 语句块则可以一次将所需要执行的 SQL 语句传输到数据库服务器，处理完毕后将数据一次传输回客户端。

2.2　PL/SQL的优势

PL/SQL 具有如下优势。

- PL/SQL 和 SQL 语言联系密切，并且被广泛使用。PL/SQL 允许使用所有的数据操纵语句、游标控制、事务控制语句，以及所有的函数、操作符和伪列，支持全部的 SQL 数据类型，并支持静态和动态 SQL。
- PL/SQL 允许将一个语句块发送至该数据库服务器，而不是单独发送 SQL 语句，从而减少了应用程序和数据库之间的网络流量。PL/SQL 子程序一旦编译就存储在数据库服务器上，处于可执行状态，并且是对用户共享，只需要客户端的一个指令就可以完成子程序的任务，不需要编译，也不需要传输子程序代码。
- PL/SQL 应用程序不受操作系统的限制，只要运行了 Oracle 数据库即可。
- PL/SQL 子程序在数据库服务器上只有一份存储，所以管理方便。
- 支持面向对象编程 OOP。
- 支持开发 Web 应用程序。

2.3　PL/SQL的语句块基本结构

任何编程语言都具有一定的语言结构，通过这种结构可编写出具有一定功能的代码块，这种代码结构也是一种逻辑结构，Oralce 的 PL/SQL 语句块的结构如图 2-3 所示。可以看到 PL/SQL 语句块由 4 部分组成，即块头区、声明区、执行区和异常区。

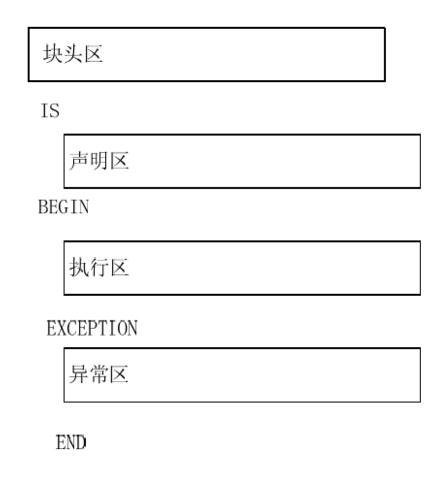


图 2-3　PL/SQL 的语句块结构

在前面的章节中已经说明，PL/SQL 语言可以编写存储过程、函数以及触发器，而这些数据库对象的创建都是使用 PL/SQL 语句块结构实现的，在本节的最后会给出一个创建存储过程的实例，下面我们详细介绍块头区、声明区、执行区和异常区。

2.3.1　块头区

块头区包含程序单元的名字和参数，其中程序单元的名字可以是函数（FUNCTION）、存储过程（PROCEDURE）或者包（PACKAGE），而参数具有一定的数据类型，该参数分为三类：一类是 IN 参数，该参数表示将参数传递给程序段单元，如存储过程；另一类是 OUT 参数，该参数返回给调用该程序（如函数）的对象；最后一类是双向的 IN OUT 参数。块头区的结构如下所示。

```
Program_type program_name （[parameter_name IN / OUT /IN OUT type specs,]……)
[RETURN datatype]
```

其中 Program_type 可以是 FUNCTION、PROCEDURE 或 PACKAGE，参数类型是 PL/SQL 定义的数据类型，而 specs 可以包含关键字 NOT NULL，以确保该参数值一定存在，如果没有参数值则使用默认值。

对于函数 FUNCTION 而言必须返回数值，函数可以在其执行部分的任意位置使用 RETURN 关键字，返回该函数定义的返回数据类型。创建 PL/SQL 函数时的块头区代码如下所示。

```
CREATE OR REPLACE FUNCTION fun_test(f float)
RETURN float
```

上述代码定义了一个函数 FUNCTION，函数名为 fun_test，传递给该函数的参数是一个浮点数 float，而该函数的返回值也是一个浮点数。

PL/SQL 过程没有返回值，创建 PL/SQL 过程（PROCEDURE）的块头区如下所示。

```
CREATE OR REPLACE PROCEDURE pro_test(name IN varchar2)
```

上述代码创建了 PL/SQL 过程 PROCEDURE，过程名为 pro_test，参数名为 name，参数类型为 varchar2。

其实也可以创建不包含块头的 PL/SQL 程序单元（如过程等），这样就可以内嵌地使用这些匿

名程序单元，这样的程序单元不好维护，且其他程序单元无法调用它，在实际应用中最好不要使用匿名块的方式。

2.3.2 声明区

声明区的目的是声明一些变量，这些变量在块内是有效的，变量的数据类型可以是任意 Oracle 定义的数据类型，如 VARCHAR2、CHAR、NCHAR、NUMBER 等。在变量声明中可以使用 CONSTRAINT 约束，从而对变量的值做出限制，如不允许为 NULL 值等。

正如在其他编程语言，如 C++或 Java 中一样，PL/SQL 语言允许使用 CONSTANT 常量来标识一个变量，该变量一旦赋值，在整个程序单元执行期间将保持不变。在变量声明时，可以使用赋值运算符“:=”为变量赋值，也可以使用 DEFAULT 关键字为变量或常量赋值。

下面给出几个实例。

- 声明一个字符变量:

```
Var_name VARCHAR2(20);
```

- 声明一个带约束的字符变量:

```
Var_name VARCHAR2(20) NOT NULL;
```

- 声明一个常量且赋初始值:

```
Var_name CONSTANT VARCHAR2(20) := 'China';
```

- 声明一个变量且使用 DEFAULT 关键字为变量赋值:

```
Var_name INTEGER DEFAULT 3.1415925;
```

PL/SQL 的声明区在块头区的后面，可使用 IS 关键字说明后面是声明区，也可以使用 DECLARE 关键字在程序单元的任意位置声明变量，代码如下所示。

```
DECLARE
   Var_name [CONSTRAINT] datatype [(constraint)] [:= value];
```

2.3.3 执行区

PL/SQL 的执行区用于完成该程序单元的功能逻辑，即该程序单元的行为定义部分，在执行部分可以使用流程控制，以及复杂的算法，是 PL/SQL 程序单元的主体部分。

执行区使用 BEGIN 和 END 标识，其中 BEGIN 标识执行区的开始，而 END 标识执行区的结束。Oracle 对执行区的要求是至少包含一条执行语句，甚至可以是 NULL，但是不能为空，并且执行区在 PL/SQL 程序单元中是必须定义的。执行区的结构如下所示。

```
BEGING
 //LOGIC STATEMENTS
END;
```

2.3.4 异常区

同任何计算机编程语言一样，有效地处理异常是该语言健壮性的体现，在 PL/SQL 语言中设计

了异常区，该区块位于 PL/SQL 块结构 END 关键字之前，用于捕获关键字 END 之前的 PL/SQL 块抛出的异常并获得处理。

在声明区中一般需要定义异常变量，在 PL/SQL 块内出现异常的地方抛出该异常，并在异常区捕获该类异常且处理，异常区的结构如下所示。

```
EXCEPTION
    WHEN exception_name1
    THEN
            Handl error1;
WHEN exception_name2
    THEN
            Handl error2;
    WHEN others
        Default error handling;
```

EXCEPTION 说明下面是一个异常区，而 WHEN 后为一个具体的异常，THEN 后是对该异常的处理代码，一个异常区可由多个 WHEN、THEN 搭配使用来处理不同的异常。

为了说明异常区的使用，下面给出实例 2-1，在该实例中声明了自定义异常。

实例 2-1　创建异常区示例程序。

```
CREATE OR REPLACE PROCEDURE pro_test(f float)
IS
    var_name varchar2(20);
//定义异常对象
    exception1 EXCETION;
BEGIN
    Statement1;
BEGING
    Statement2;
//抛出在执行 Statement2 语句时的异常
Raise exception1;
    EXCEPTION
    WHEN exception1
Handling errors;
END;
Statement3;
//在 statement1 和 statement3 中抛出的异常在下面这个异常区中处理
EXCEPTION
    WHEN OTHERS
        handling errors;
END;
```

下面详细说明上例中异常的执行过程：执行 Statement1，如果此时发生异常，则在最后的异常处理区块中处理，接着执行执行区内的一个 BEGIN…END 区块，如果 Statement2 发生异常，则抛出异常 exception1，而该异常在随后的异常处理区块中处理，接着执行 Statement3，如果此时发生异常，则在该语句后的 EXCEPTION 区块中处理。

2.4　PL/SQL的语句块执行过程

在上一节中我们已经讨论了 PL/SQL 语句块的基本结构，这些结构包括块头部分、声明部分、

可执行部分以及异常处理部分，现在我们通过一个实例来演示如何将这些部分组合起来。

为了显示输出，先执行如下指令。

```
SQL> set serveroutput on;
```

下面编写并执行一个 PL/SQL 块，该程序段的作用是设置两个变量，然后将从 employees 表中读取的数据存入这两个变量，并输出显示，如实例 2-2 所示。

实例 2-2　理解 PL/SQL 语句块。

```
SQL>DECLARE
  2    var_first_name varchar2(30);
  3    var_last_name  varchar2(30);
  4  BEGIN
  5    SELECT first_name,last_name
  6      INTO var_first_name,var_last_name
  7      FROM employees
  8     WHERE employee_id=168;
  9    DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
 10    DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
 11  EXCEPTION
 12    WHEN NO_DATA_FOUND THEN
 13    DBMS_OUTPUT.PUT_LINE('no data find');
 14* END;

var_first_name is : Lisa
var_last_name is : Ozer

PL/SQL procedure successfully completed.
```

以上 PL/SQL 语句块包含 3 个部分。

第 1 部分为声明部分。

```
DECLARE
      var_first_name varchar2(30);
      var_last_name  varchar2(30);
```

这部分声明了两个变量：var_first_name 和 var_last_name，数据类型都为 varchar2。

第 2 部分为执行部分。

```
4  BEGIN
5    SELECT first_name,last_name
6    INTO var_first_name,var_last_name
7    FROM employees
8    WHERE employee_id=168;
9    DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
10   DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
11  END;
```

这部分是 PL/SQL 语句块的核心，也是必不可少的部分，通过 SQL 语句 SELECT 从表 employees 中读出 employee_id=168 的员工的 first_name 和 last_name，并分别存储到两个变量 var_first_name 和 var_last_name 中，最后输出这些变量的值。

第 3 部分为异常部分。

```
  4  BEGIN
  5    SELECT first_name,last_name
```

```
  6       INTO var_first_name,var_last_name
  7       FROM employees
  8      WHERE employee_id=168;
  9    DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
 10    DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
 11  EXCEPTION
 12    WHEN NO_DATA_FOUND THEN
 13    DBMS_OUTPUT.PUT_LINE('no data find');
 14* END;
```

在可执行部分我们加入了异常处理，这个异常是 Oracle 定义的异常，由异常条件与处理语句组成，如果发生异常，则程序执行过程进入 EXCEPTION 部分，用于判断异常类型是否是 NO_DATA_FOUND，如果是，则在屏幕打印输出'no data find'。

例如修改第 8 行 employee_id 的值为 16，则执行代码如实例 2-3 所示。

实例 2-3　修改示例。

```
SQL>DECLARE
  2    var_first_name varchar2(30);
  3    var_last_name  varchar2(30);
  4  BEGIN
  5    SELECT first_name,last_name
  6       INTO var_first_name,var_last_name
  7       FROM employees
  8      WHERE employee_id=16;
  9    DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
 10    DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
 11  EXCEPTION
 12    WHEN NO_DATA_FOUND THEN
 13    DBMS_OUTPUT.PUT_LINE('no data find');
 14* END;

no data find

PL/SQL procedure successfully completed.
```

因为 employee_id=16 的员工没有记录，所以在 PL/SQL 代码的执行阶段发生异常，程序流程进入 EXCEPTION 部分，此时处理该异常，发现异常类型匹配，则执行如下语句。

```
DBMS_OUTPUT.PUT_LINE('no data find');
```

整个 PL/SQL 块的执行需要经历 3 个阶段，即语法检查、替代和生成伪代码。对于匿名 PL/SQL 语句块，每当执行 PL/SQL 语句块时，整个代码发送到 PL/SQL 引擎进行处理并执行编译。PL/SQL 语句块在创建或者被修改时会执行编译工作。下面我们详细介绍 PL/SQL 语句块编译的 3 个步骤。

1．语法检查

在这个阶段主要检查 PL/SQL 语句块中代码的语法格式是否合法，以及是否遵守 PL/SQL 的编程规则等，例如，我们省略了第 3 行的分号“；”，错误提示如实例 2-4 所示。

实例 2-4　PL/SQL 语句块在语法检查阶段的错误。

```
SQL>DECLARE
  2    var_first_name varchar2(30);
  3    var_last_name  varchar2(30)          //缺少分号
```

```
  4  BEGIN
  5    SELECT first_name,last_name
  6      INTO var_first_name,var_last_name
  7      FROM employees
  8     WHERE employee_id=168;
  9    DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
 10    DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
 11  EXCEPTION
 12    WHEN NO_DATA_FOUND THEN
 13    DBMS_OUTPUT.PUT_LINE('no data find');
 14* END;

BEGIN
*
ERROR at line 4:
ORA-06550: line 4, column 1:
PLS-00103: Encountered the symbol "BEGIN" when expecting one of the following:
:= ; not null default character
The symbol ";" was substituted for "BEGIN" to continue.
```

2. 替代

在语法检查完毕后，将进入替代阶段，即编译器会为变量分配存储空间用于保存数据，这样程序运行时就可以使用这个地址，并检查 PL/SQL 语句涉及的数据库对象是否存在，如将 employees 表名改写为 employee，错误提示如实例 2-5 所示。

实例 2-5　替代阶段的错误示例。

```
SQL>DECLARE
  2    var_first_name varchar2(30);
  3    var_last_name  varchar2(30);
  4  BEGIN
  5    SELECT first_name,last_name
  6      INTO var_first_name,var_last_name
  7      FROM employee           //该表不存在
  8     WHERE employee_id=168;
  9    DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
 10    DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
 11  EXCEPTION
 12    WHEN NO_DATA_FOUND THEN
 13    DBMS_OUTPUT.PUT_LINE('no data find');
 14* END;

     FROM employee
          *
ERROR at line 7:
ORA-06550: line 7, column 12:
PL/SQL: ORA-00942: table or view does not exist
ORA-06550: line 5, column 4:
PL/SQL: SQL Statement ignored
```

上述提示说明了错误原因，即表对象不存在，这个过程就是在替代阶段完成的。这个错误是一种编译错误，还有一种是运行错误，这种错误在编译阶段无法发现，如果我们选择的数据表在对象中不存在，而我们也没有异常处理代码，则会提示运行错误，这个错误是在程序被运行时发现的，如实例 2-6 所示。

实例 2-6　提示运行错误示例。

```
SQL>DECLARE
  2     var_first_name varchar2(30);
  3     var_last_name  varchar2(30);
  4  BEGIN
  5     SELECT first_name,last_name
  6        INTO var_first_name,var_last_name
  7        FROM employees
  8       WHERE employee_id=16;
  9     DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
 10     DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
 11* END;

DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5
```

如果这段代码是用于为 PL/SQL 代码命名，例如过程，此时编译该过程时不会发生错误，但是当执行该过程时则会报错，错误提示如上例所示。

3．生成伪代码

在上述两个阶段完成后将进入伪代码的生成阶段，伪代码由 PL/SQL 引擎的指令组成，命名语句块的伪代码存储在数据库服务器中，当被调用时，该程序就会运行。编译成功后，命名 PL/SQL 语句块变为 VALID 状态。而对于匿名 PL/SQL 语句块则无法保存在数据库中。

2.5　PL/SQL与SQL的区别

SQL*Plus 执行 SQL 语句与执行 PL/SQL 语句是有区别的，主要体现在结束与执行方式、显示方式上。下面通过实例更直观地说明一下这种差别。首先连接到数据库，使用 HR 模式，如实例 2-7 所示。

实例 2-7　执行查询语句。

```
SQL> select first_name,last_name from employees where employee_id=168;

FIRST_NAME           LAST_NAME
-------------------------------- -------------------------
Lisa                   Ozer
```

下面分析一下上例中 SQL 语句的执行过程，在 SQL*Plus 提示符下输入 SQL 语句，分号表示语句的结束，分号之前的部分就是一条语句，按回车键后该 SQL 语句被发送到数据库服务器执行并返回结果。

下面这个 PL/SQL 程序，与 SQL 语句的执行有些区别。对于 PL/SQL 程序，分号表示语句的结束，而使用“.”号表示整个语句块的结束，也可以省略。按回车键之后，该语句块不会执行，即不会发送到数据库服务器，而是必须使用“/”符号执行 PL/SQL 语句块，如实例 2-8 所示。

实例 2-8　PL/SQL 语句块。

```
SQL> DECLARE
  2  var_first_name varchar2(30);
  3  var_last_name  varchar2(30);
  4   BEGIN
  5  SELECT first_name,last_name
  6  INTO var_first_name,var_last_name
  7  FROM employees
  8  WHERE employee_id=168;
  9  DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
 10  DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
 11  END;
 12  .
 13  /

PL/SQL procedure successfully completed.
```

SQL*Plus 工具可以执行 PL/SQL 语句，分号"；"表示 SQL 语句的结束，"/"符号表示执行语句，此时会发送 SQL 语句到数据库服务器进行处理。

读者或许会注意到在上例中，虽然执行了 PL/SQL 程序，但是没有输出结果的显示。数据库服务器肯定将数据传输给了 SQL*Plus，并且变量 var_fisrt_name 和 var_last_name 都已经被赋值，问题是并没有显示在 SQL*Plus 上，于是执行如下指令：

```
SQL> set serveroutput on;
```

该输出显示在默认情况下是关闭的，为了方便，可以编写一个触发器，一旦数据库登录则修改该参数。

2.6　DBMS_OUTPUT包的使用

在我们编写的 PL/SQL 实例中，多次使用包 DBMS_OUTPUT，我们调用了该包的一个过程 PUT_LINE，从而将数据库服务器返回的数据显示在 SQL*Plus 上面。

DBMS_OUTPUT.PUT_LINE 调用了过程 PUT_LINE，其作用是把数据库服务器返回的数据缓存到内存，一旦 PL/SQL 语句执行结束就显示在屏幕上，所以在实现该功能之前需要执行一条设置语句。

```
SQL> set serveroutput on;
```

此时，读者应该理解了为什么要设置该参数为 ON 了。既然数据可以缓存在内存，那么对于缓存的大小就应该可以控制。我们可以使用如下指令设置缓存的大小：

```
SQL> set serveroutput on size 4000;
```

如果不需要将 PL/SQL 程序返回的数据显示在屏幕上，则可以关闭这个数据缓存并显示功能。关闭该功能的代码如下所示：

```
SQL> set serveroutput off;
```

DBMS_OUTPUT 包主要起到用户调试的目的，该包的存储过程 PUT_LINE 将数据库服务器返回的数据存放到缓存中，这样这些数据就可以显示在屏幕上，并且可以被触发器、其他存储过程以

及程序包读取。DBMS_OUTPUT 包的子程序概要如表 2-1 所示。

表 2-1　DBMS_OUTPUT 包的子程序说明

DBMS_OUTPUT 子程序	功能
DISABLE 存储过程	禁止消息输出
ENABLE 存储过程	启用消息输出
GET_LINE 存储过程	从 BUFFER 获取单行数据
GET_LINES 存储过程	从 BUFFER 中获取多行数据
NET_LINE 存储过程	输出一个换行，没有参数
PUT 存储过程	将一行数据放入 BUFFER 中
PUT_LINE 存储过程	将多行数据放入 BUFFER 中

2.7　替代变量的使用

下面来讨论使用替代变量的优势，从编程的角度来讲，使用替代变量可以使得程序更加灵活，替代变量可以接收用户的输入信息，SQL 语句发送到数据库服务器之前将绑定一个具体的值，然后再发送过去执行，未使用替代变量的实例如实例 2-9 所示。

实例 2-9　未使用替代变量的实例。

```
DECLARE
       var_first_name varchar2(30);
       var_last_name  varchar2(30);
    BEGIN
       SELECT first_name,last_name
          INTO var_first_name,var_last_name
          FROM employees
         WHERE employee_id=168;
       DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
       DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
   END;
.
/
```

在上例中，我们查询了表 employees 中 employee_id 为 168 的员工信息，这个匿名 PL/SQL 块功能单一。此时如果我们想扩展该程序的灵活性，例如查询所有 employee_id 的员工信息，我们希望按照需求输入 ID 号，此时就可以考虑使用替代变量。替代变量的符号是“&”或者“&&”，下面我们改写上例，如实例 2-10 所示。

实例 2-10　使用替代变量的实例。

```
DECLARE
       var_first_name varchar2(30);
       var_last_name  varchar2(30);
       var_employee_id number :=&b_employee_id;
    BEGIN
       SELECT first_name,last_name
          INTO var_first_name,var_last_name
          FROM employees
```

```
        WHERE employee_id=var_employee_id;
       DBMS_OUTPUT.PUT_LINE('var_first_name is : '||var_first_name);
       DBMS_OUTPUT.PUT_LINE('var_last_name is : '||var_last_name);
    END;
  .
  /
```

在 DECLARE 部分定义了一个变量 var_employee_id，数据类型为 number 并且使用替代变量。在 WHERE 中使用替代变量执行该 PL/SQL 程序，输出如下所示。

```
SQL> /
Enter value for b_employee_id: 168
old   4:         var_employee_id number :=&b_employee_id;
new   4:         var_employee_id number :=168;
var_first_name is : Lisa
var_last_name is : Ozer

PL/SQL procedure successfully completed.
```

在执行该 PL/SQL 程序时，首先要求输入替代变量的值，并且程序会输出 OLD 值和 NEW 值。NEW 值替代了变量的值，实际传输是该替代后的语句，然后整个 PL/SQL 块被传送到数据库服务器执行，如果执行成功就返回数据，如以上代码所示。如果该员工不存在，则输出错误提示，如以下代码所示。

```
SQL> /
Enter value for b_employee_id: 1000
old   4:         var_employee_id number :=&b_employee_id;
new   4:         var_employee_id number :=1000;
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 6
```

如果不希望看到如下两行数据输出，可以使用 SET VERIFY OFF 将其关闭。

```
old   4:         var_employee_id number :=&b_employee_id;
new   4:         var_employee_id number :=168;
```

下面我们关闭这个显示功能，再次执行。

```
SQL> set verify off;
SQL> /
Enter value for b_employee_id: 168
var_first_name is : Lisa
var_last_name is : Ozer

PL/SQL procedure successfully completed.
```

我们在讲解替代变量时已经讲过替代变量可以以“&”开头，也可以使用“&&”开头，那么使用“&&”开头的替代变量显然与“&”开头的替代变量有所区别，实际上使用“&&”开头的替代变量说明只要当前 PL/SQL 块中定义相同的替代变量，在执行该 PL/SQL 块时，只需要输入一次该替代变量的值。下面给出一个实例，此时我们使用“&”开头的替代变量定义两个名称相同的替代变量，如实例 2-11 所示。

实例 2-11　使用“&”开头的替代变量。

```
 SET SERVEROUTPUT ON;
BEGIN
   DBMS_OUTPUT.PUT_LINE('TOM is from : ' ||'&country');
   DBMS_OUTPUT.PUT_LINE('LARRY is from : ' ||'&country');
END;
.
/
```

上例中，我们使用了替代变量&country，两个替代变量定义相同，下面是该 PL/SQL 匿名块的执行结果。

```
Enter value for country: England
old   2:    DBMS_OUTPUT.PUT_LINE('TOM is from : ' ||'&country');
new   2:    DBMS_OUTPUT.PUT_LINE('TOM is from : ' ||'England');
Enter value for country: America
old   3:    DBMS_OUTPUT.PUT_LINE('LARRY is from : ' ||'&country');
new   3:    DBMS_OUTPUT.PUT_LINE('LARRY is from : ' ||'American;');
TOM is from : England
LARRY is from : America

PL/SQL procedure successfully completed.
```

在执行 PL/SQL 块时，编译器遇到第 1 个替代变量，要求输入该变量的值，我们输入 England，而后遇到第 2 个替代变量，同样要求输入该变量的值，我们输入 America。此时编译器将这两个变量赋予具体的值，因为不涉及 SQL 语句，此时 PL/SQL 引擎处理该语句并输出数据。我们看到在这个实例中，对于定义相同的替代变量输入了两次值。那么是否可以只输入一次值呢？这时就要使用以“&&”开头定义的替代变量。在 PL/SQL 块中的多个相同的替代变量中，将第 1 个出现的替代变量修改为以“&&”开头，如实例 2-12 所示。

实例 2-12　使用“&&”开头的替代变量。

```
SET SERVEROUTPUT ON;
BEGIN
   DBMS_OUTPUT.PUT_LINE('TOM is from : ' ||'&&country');
   DBMS_OUTPUT.PUT_LINE('LARRY is from : ' ||'&country');
END;
.
/
```

再次执行上述代码，看一下是否可以输入一次替代变量的值就可以执行完该语句。下面是该语句的执行结果。

```
Enter value for country: England
old   2:    DBMS_OUTPUT.PUT_LINE('TOM is from : ' ||'&&country');
new   2:    DBMS_OUTPUT.PUT_LINE('TOM is from : ' ||'England');
old   3:    DBMS_OUTPUT.PUT_LINE('LARRY is from : ' ||'&country');
new   3:    DBMS_OUTPUT.PUT_LINE('LARRY is from : ' ||'England');
TOM is from : England
LARRY is from : England

PL/SQL procedure successfully completed.
```

此时，只要求输入一次替代变量&country 的值，则在整个 PL/SQL 块中，所有相同的替代变

量都可以使用这个值。

2.8　本章小结

本章首先介绍了 PL/SQL 的应用环境以及 PL/SQL 的优势，PL/SQL 与 SQL 都需要引擎处理，各司其职，协同完成 PL/SQL 的编译工作，然后介绍了 PL/SQL 的基本结构以及 PL/SQL 语句块的执行过程，在理解 PL/SQL 语句块的执行过程后将有利于处理 PL/SQL 的编程错误，PL/SQL 与 SQL 的执行具有明显区别，在编程时要注意这个变化。最后介绍了 DBMS_OUTPUT 包以及替代变量。

第 3 章

数据库管理工具SQL*Plus

SQL*Plus 是 Oracle 数据库管理系统提供的一个工具软件，其提供一个人机接口，通过 SQL*Plus 可管理和维护数据库，如常用的查询数据表信息、系统信息、数据文件等，它提供了一系列指令，通过这些指令可以简化用户的指令或者格式化输出信息。还提供了编写脚本文件的功能，可以极大地提高 DBA 管理数据库的效率。SQL*Plus 作为数据库管理工具可以设置友好的环境变量，以方便 DBA 的管理和需求维护。

3.1 SQL*Plus的启动

SQL*Plus 是一个工具软件，可在其中输入 SQL 语句或者 SQL*Plus 的指令，前者用于数据库操作，如数据查询、更改和删除等，后者主要用于设置 SQL*Plus 的环境变量，也提供了一些方便 DBA 输入和编辑 SQL 指令的方式。值得一提的是在 Oracle 9i 及 10g 版本中，Oracle 提供了一个 Web 版本的 SQL*Plus 工具，即 iSQL*Plus。本节将讲述如何启动和使用 SQL*Plus、iSQL*Plus。在 Oracle 9i、Oracle 10g 中启动 SQL*Plus 的方式基本相同，显示界面也相同，而在 Oracle 11g 中启动 SQL*Plus 工具的显示界面区别很大，下面我们先介绍在不同环境下启动 SQL*Plus 的方式，这样更有利于说明一些概念。

3.1.1 启动 SQL*Plus 工具

Oracle 在安装数据库管理系统时，无论是服务器端还是客户端，都默认安装了 SQL*Plus 工具，可以按照如下步骤打开 SQL*Plus 工具，在 Oracle9i 以及 10g 中会提供一个 SQL*Plus 用户界面，可通过这个用户界面来登录数据库，并维护和管理数据库，在 Oracle 11g 中不再提供这个接口，而是打开一个类似于 DOS 的界面，不过登录数据库的过程类似。

1. 在 Oracle 9i 和 10g 中启动 SQL*Plus

依次选择【开始】→【程序】→【Oracle-OraHome90】→【Application Development】→【SQL*Plus】，如图 3-1 所示，单击 SQL*Plus 工具即可启动该管理软件。

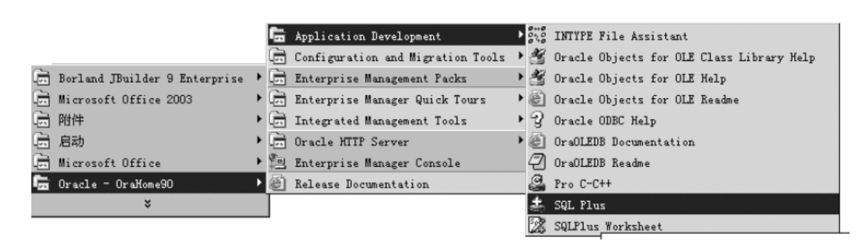


图 3-1　打开 SQL*Plus 工具

此时会弹出如图 3-2 所示的用户界面，注意此时需要输入三个参数：用户名、口令和主机字符串。如果用户想登录默认数据库或者只安装了一个数据库可以不填写，采用系统默认值即可，否则需要填写主机字符串。

在图 3-2 中输入用户名 scott，密码 tiger，单击“确定”按钮或者直接按回车键，即可打开如图 3-3 所示的用户界面。在该界面中，鼠标停在“SQL>”后，此时就可以输入 SQL 查询指令或者设置 SQL*Plus 的环境变量了。从该图中可以看到关于该工具的信息：如 SQL*Plus 的版本号为“9.0.1.0.1”，此次登录的时间是“星期四 5 月 14 10:52:21 2009”；该数据库为 Oracle 9i 企业版，即 Oracle 9i Enterprise Edition Release 9.0.1.1.1 - Production 等。

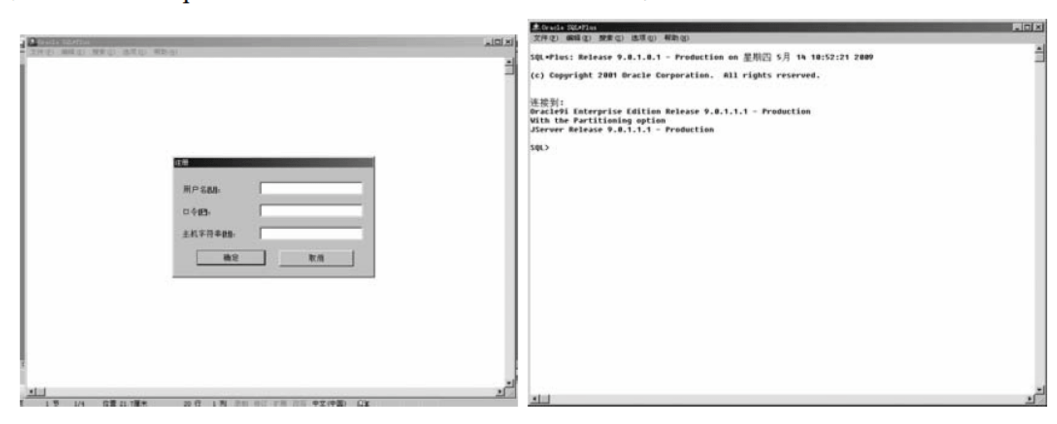


图 3-2　登录 SQL*Plus　　　　图 3-3　打开的 SQL*Plus 工具



scott 是 Oracle 数据库创建的默认用户，密码为 tiger，scott 为 Oracle 公司创建初期一员优秀的程序员，但是他曾经使用 C 语言编写关系数据库管理系统，密码则来自他养的一只猫，因为他亲切的称呼这只猫为 tiger，所以就使用其作为密码，或许是为了让人们记住这位优秀的程序员吧，Oracle 数据库一只保留这个默认的用户名和密码。

读者可以尝试在图 3-3 的“SQL>”符号后输入 SELECT * FROM dept 指令，查看结果并体验该工具的作用，在语句的结尾一定要有英文输入法的分号“;”作为查询语句的结束。该语句的作用是在当前用户模式下查找表 dept 内的全部信息，“*”号是通配符，表示查找表 dept 中所有列的信息，而表 dept 是用户 scott（应该还记得刚才登录时使用的用户名吧）所拥有的表，查询方法与查询结果如图 3-4 所示。

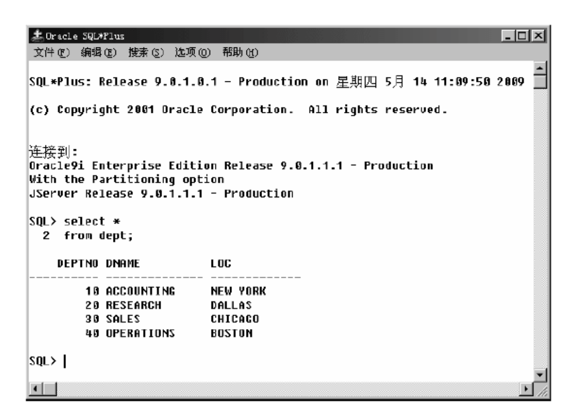


图 3-4　使用 SQL*Plus 查询表 dept 的数据



在输入多行 SQL 语句时，SQL*Plus 会自动添加行号。通过输入行号，可以设置该 SQL 语句为当前行。

其实，利用 Oracle 安装软件目录中的工具启动 SQL*Plus，在不同版本的数据库中操作略有不同，我们还是推荐在 DOS 下使用“SQLPlus /NOLOG”指令启动 SQL*Plus，然后连接数据库服务器。

2. 在 Oracle 11g 中启动 SQL*Plus

在 Oracle 11g 中不会再打开一个单独设计的 Oracle SQL*Plus 对话框，如图 3-2 所示，而是打开一个类 DOS 的对话窗口，下面介绍在 Oracle 11g 中如何启动 SQL*Plus 工具，使用 scott 用户名登录数据库，并执行如图 3-5 所示的查询。

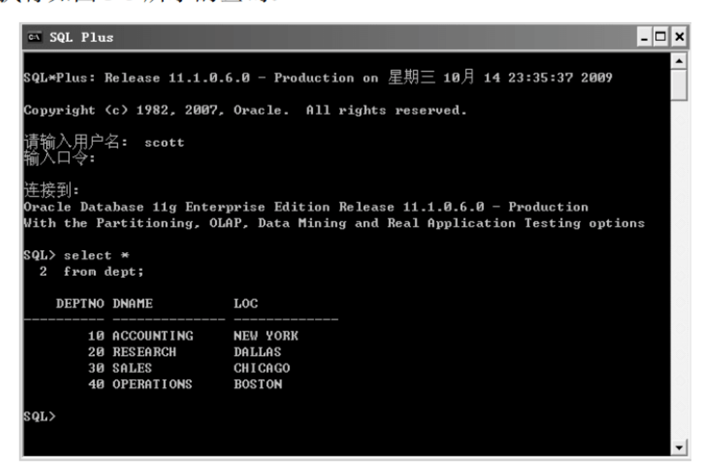


图 3-5　登录 SQL*Plus

在图 3-5 的最上部分，显示了数据库的版本以及打开 SQL*Plus 的时间，在输入用户名 scott 和密码 tiger 后，按回车键即可连接到 Oracle 11g 数据库，此时进入“SQL>”状态，可以执行 SQL 指令，在这里查询用户 scott 下的表 dept 的信息，这个查询结果表明，此时可以执行 scott 用户权限的数据操作了。

3.1.2 启动 iSQL*Plus 工具

虽然在 Oracle 11g 中已经不支持 iSQL*Plus 工具，但是该工具在 Oracle 9i 以及 10g 版本中依然可以使用，所以在这里依然简单介绍一下如何启动和使用这个基于六拉尼的管理工具。

经过上节的讲述，读者已经理解了如何利用 C-S 模式远程管理数据库、在客户端启动 SQL*Plus 工具、输入主机字符串来连接要管理的数据库。这种模式的局限在于必须在计算机上安装 Oracle 客户端软件，要求使用 Net Manager（不同版本的名字略有不同）配置网络连接参数。

显然这种方式有些烦琐，如果可以使用浏览器直接登录数据库服务器就会很方便，在 Oracle 9i 及 10g 版本中，提供了一个类似于 SQL*Plus 的 iSQL*Plus 工具，它是 SQL*Plus 的 Web 版本，可使用任意类型的浏览器连接数据库服务器，“i”表示它是一个网络管理工具。

在 Oracle 9i 中启动该工具前，首先要启动 Apache 支持的 iSQL*Plus 服务器，如图 3-6 所示。

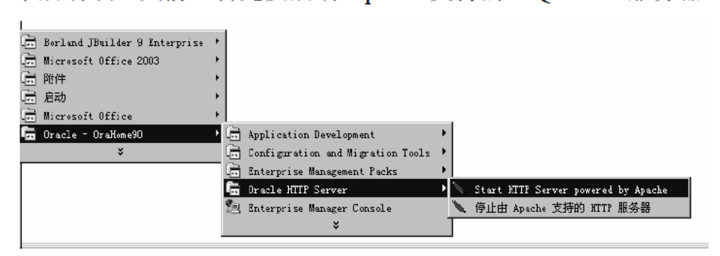


图 3-6 启动 Apache 服务器

然后打开浏览器，在地址栏中输入 http://localhost/isqlplus，打开如图 3-7 所示的登录界面。

在图 3-7 中输入用户名、密码、连接标识符、选择用户权限。这里笔者使用 scott 用户名，密码为 tiger，因为是登录默认的数据库服务器，也是本机唯一的服务器，所以不输入主机字符串，权限选择用户（scott 用户是普通用户，如果选择其他选项，如 AS SYSDBA 或者 AS SYSOPER 都会无法正常登录），至于其他两个权限，将在后面进行详细介绍，这里不再赘述。

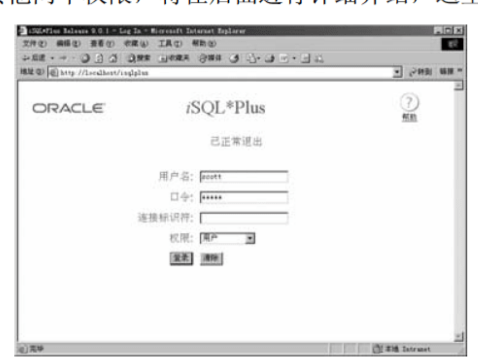


图 3-7 iSQL*Plus 登录界面

单击“登录”按钮或者按下回车键，将会显示如图 3-8 所示的 iSQL*Plus 界面。在界面的左下角会看到“已连接”字样。在“输入语句”列表框中可以输入 SQL 语句。

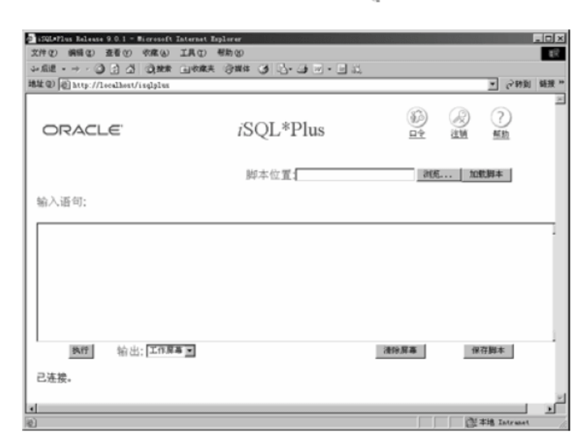


图 3-8　iSQL*Plus 界面

为了更直观地体验 iSQL*Plus 的使用，可在“输入语句”列表框中输入 SQL 语句，如图 3-9 所示。



图 3-9　执行 SQL 语句



在执行 SQL 语句时，必须单击“执行”按钮，“输出”选项可以选择数据输出的位置，共有 3 个选项：工作屏幕，即在当前窗口显示；文件，即保存到指定目录下，自己输入文件名；窗口，即在新窗口中显示输出数据。

对于 iSQL*Plus 的具体功能不再一一演示，这里对它的工作做一下概括性的介绍，如果读者需要使用该工具，可以根据以下介绍的功能去参考其他资料。

- 可使用 SQL 指令查询、修改和删除数据库中的数据。
- 数据行的显示更加多样化，可以在当前窗口或者新窗口中把数据行存储为文件的形式（使用浏览器打开）。
- 可创建脚本文件。
- 可更改口令，并具有用户切换功能。

3.2 SQL*Plus的常用指令

在使用 SQL*Plus 工具时，往往需要对输入的 SQL 语句进行操作，如对刚执行的指令进行修改以减少再输入全部指令的时间；格式化列的名称，使得输出更加人性化，或者对每列数据的显示宽度进行调整，使得表的输出更加规范化等，本节主要讲述 SQL*Plus 提供的常用指令，并用实例演示这些指令的作用以及如何应用等。

3.2.1 desc 指令

desc 指令是指令 description 的缩写。在查询表中的数据时，如果我们对表的属性不了解，此时只要知道表的名字，就可以使用该指令获得该表的列属性信息。我们需要再次说明 scott 用户是在创建数据库时系统默认设置的一个用户，该用户具有一定的权限和一些表，读者可以通过这些表练习 SQL 语句。

下面给出实例 3-1，用于查询 scott 用户中表 dept（部门表）的属性信息。

实例 3-1　查询 scott 用户中表 dept 的属性信息

```
C:\>sqlplus /nolog

SQL*Plus: Release 10.2.0.1.0 - Production on 星期六 12 月 17 20:03:59 2011

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

SQL> connect scott/oracle
已连接。
SQL> desc dept;
 名称                                      是否为空? 类型
 ----------------------------------------- -------------------------------
 DEPTNO                                     NOT NULL NUMBER(2)
 DNAME                                      VARCHAR2(14)
 LOC                                         VARCHAR2(13)
```

这里的表结构是指表中的数据具有哪些属性，这些属性就是表中列的名字，如例 3-2 中，表 dept 由 3 列组成，第 1 列为 DEPTNO（部门号），第 2 列为 DNAME（部门名），第 3 列为 LOC（部门所在地）。

其中：

- 第 1 列 DEPTNO 的数据类型为整数，最大长度为 2。
- 第 2 列 DNAME 的数据类型为变长字符型，最大长度为 14 个字符。
- 第 3 列 LOC 的数据类型为变长字符型，最大长度为 13 个字符。



在实例 3-1 中给出了在 DOS 下启动 SQL*Plus 的方法，以及使用 SQL*Plus 工具和 scott 用户连接数据库的方法。connect 指令用于连接数据库，基本语法为 connect usrname/password@sid，因为是连接本机默认的数据库，所以可不使用服务名。

3.2.2　column 指令

顾名思义，column 是与列相关的指令。通过对列的输出值和列本身进行适当的格式化，使表数据显得更加人性化，也可以说，column 指令使得 DBA 更容易通过 SQL*Plus 得到显示更加合理的表。首先给出该指令的语法格式。

```
col[umn] [{column|expr}[option...]]
```

这里的[option]包括如下参数：

- FOR[MAT] format
- CLE[AR]
- HEA[DING]text
- JUS[TIFY] {L[EFT]|C[ENTER] |R[IGHT]}
- NEWL[INE]
- NOPRI[NT]|PRI[NT]
- NUL[L]text
- ON|OFF

下面依次介绍这些指令的用法，主要通过实例使得读者可以直观地理解如何使用 column 指令。

1. FOR[MAT] format

首先查询表 salgrade 的全部信息。

（1）格式化模式 ‘9’

查看表 salgrade 中的全部数据，如实例 3-2 所示。

实例 3-2　查看表 salgrade 中的全部数据。

```
SQL> SELECT *
  2  FROM salgrade;

     GRADE   LOSAL      HISAL
--------------- -------------- -----------------
         1       700       1200
         2      1201       1400
         3      1401       2000
         4      2001       3000
         5      3001       9999
         1       700       1200
         2      1201       1400
         3      1401       2000
         4      2001       3000
         5      3001       9999
```

显然该表中的三列（RADE、LOSAL 和 HISAL）都占用了过多的宽度，使用如下的 column 指令格式化这些列的输出，使其占用较少的字符宽度，如实例 3-3 所示。

实例 3-3　格式化 number 类型数据（不带小数点）。

```
SQL> col grade for 99
```

```
SQL> col losal for 9999
SQL> col hisal for 9999
```

再次执行实例 3-3：

```
SQL> SELECT *
  2  FROM salgrad

GRADE LOSAL HISAL
---------- ---------- --------
      1   700  1200
      2  1201  1400
      3  1401  2000
      4  2001  3000
      5  3001  9999
      1   700  1200
      2  1201  1400
      3  1401  2000
      4  2001  3000
      5  3001  9999
```

显然该表中的三列：GRADE、LOSAL 和 HISAL 占用的宽度明显缩小了，即使用 column 指令格式化这些列的输出，可使其占用较少的字符宽度。其中 GRADE 列占用 2 位数字的宽度，而 LOSAL 和 HISAL 各自占用 4 位数字的宽度。由于这 3 列的数据类型都为数字类型（NUMBER），所以使用“9”这样的选项，99 是格式化模式，每个“9”表示一位数字。

但是这里没有小数点，也不显示 0，那如何做到既显示小数点，又显示 0 呢？读者可以通过实例 3-4 体会其方法。

实例 3-4　格式化 NUMBER 类型数据（带小数点）。

```
SQL> col hisal for 9999.99
SQL> SELECT *
  2  FROM salgrade;

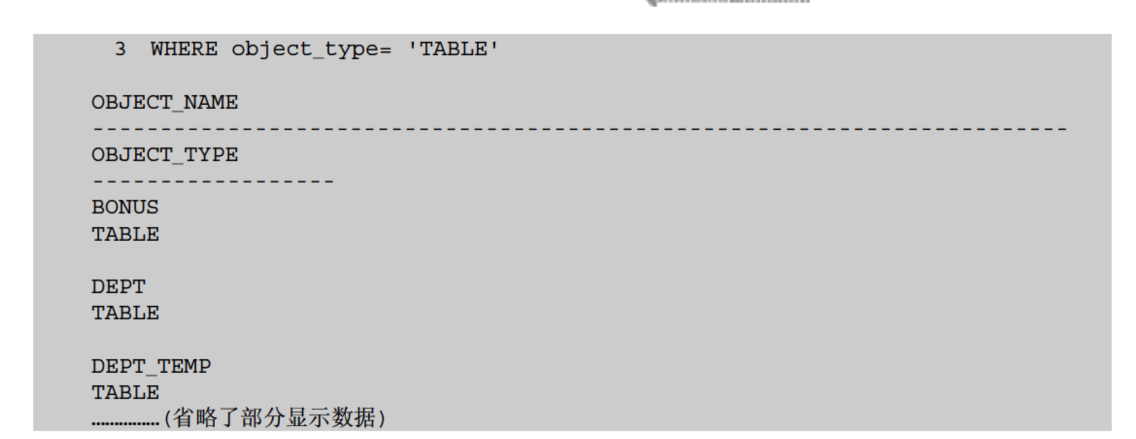
GRADE LOSAL    HISAL
---------- ---------- -------------
      1   700  1200.00
      2  1201  1400.00
      3  1401  2000.00
      4  2001  3000.00
      5  3001  9999.00
      1   700  1200.00
      2  1201  1400.00
      3  1401  2000.00
      4  2001  3000.00
      5  3001  9999.00
```

（2）格式化模式 ‘a’

如果显示结果的某列是字符型的，且占用较大宽度，此时可以使用格式化模式 ‘a’，首先，我们执行测试，如实例 3-5 所示。

实例 3-5　查看表 user_objects 中的部分信息。

```
SQL>SELECT object_name ,object_type
  2  FROM user_objects
```



```
  3  WHERE object_type= 'TABLE'

OBJECT_NAME
--------------------------------------------------------------------------
OBJECT_TYPE
------------------
BONUS
TABLE

DEPT
TABLE

DEPT_TEMP
TABLE
..............（省略了部分显示数据）
```

分析：从输出结果可以看出，列 OBJECT_NAME 占用了较多的字符宽度，使得显示的数据很不协调，可以采用如实例 3-6 所示的 column 指令将其格式化为 20 个字符宽度。

实例 3-6　格式化字符类型数据。

```
SQL> col object_name for a20
```

重新执行查询指令，如实例 3-7 所示。

实例 3-7　格式化后再次查询 user_objects 中的部分数据。

```
SQL>SELECT object_name ,object_type
  2  FROM user_objects
  3  WHERE object_type= 'TABLE'

OBJECT_NAME          OBJECT_TYPE
-------------------------------------  ----------------------------------
BONUS                TABLE
DEPT                  TABLE
DEPT_TEMP            TABLE
EMP                   TABLE
EMP_TEMP             TABLE
ORD                   TABLE
PRODUCT              TABLE
SALGRADE             TABLE
SUPPLIER              TABLE
```

从输出可以看出，列 OBJECT_NAME 的宽度缩小了，改为 20 个字符的宽度，正因为该列是字符型数据，所以采用“a20”来格式化，表示把列的输出格式化为 20 个字符宽度。

（3）格式化模式 ‘$’

实例 3-4 中的工资数据不是很合理，不知道是美元、英镑还是人民币，采用“$”格式化可以解决这个问题，如实例 3-8 所示。

实例 3-8　格式化货币。

```
SQL> col losal for $9999
SQL> col hisal for $9999
```

通过上述格式化操作，再使用 SQL 语句查询，结果显示表的第 2 列的数据头部添加了“$”

符号，如实例 3-9 所示。

实例 3-9　格式化货币后查询表 salgrade 中的所有数据。

```
SQL> SELECT *
  2  FROM salgrade;

     GRADE  LOSAL      HISAL
---------- ------ ----------
         1   $700      $1200
         2  $1201      $1400
         3  $1401      $2000
         4  $2001      $3000
         5  $3001      $9999
         1   $700      $1200
         2  $1201      $1400
         3  $1401      $2000
         4  $2001      $3000
         5  $3001      $9999
```

（4）格式化模式 ‘L’

如果数据库安装在不同的国家，且希望显示本地的货币格式，那应该如何处理呢？SQL*Plus 使用了 L 格式化模式来解决这个问题。将实例 3-9 中表的第 2 列和第 3 列都改为本地货币格式。输入 column 指令如实例 3-10 所示。

实例 3-10　本地货币格式化。

```
SQL> col losal for L9999
SQL> col hisal for L9999
```

再次查询表 salgrade 的所有数据，如实例 3-11 所示。

实例 3-11　本地货币格式化后查询表 salgrade 的所有数据。

```
SQL> SELECT *
  2  FROM salgrade;
   GRADE           LOSAL           HISAL
-------------- ---------------------------- ----------------------
         1          RMB700       RMB1200
         2         RMB1201       RMB1400
         3         RMB1401       RMB2000
         4         RMB2001       RMB3000
         5         RMB3001       RMB9999
         1          RMB700       RMB1200
         2         RMB1201       RMB1400
         3         RMB1401       RMB2000
         4         RMB2001       RMB3000
         5         RMB3001       RMB9999
```

从输出结果可以看出，此时的货币单位是人民币。这是因为 column 命令的格式化模式“L”是显示本地货币的。其实，它是根据 Oracle 数据库的字符集来确定的，因为我们安装的数据库的字符集为中文，所以显示的本地货币为人民币（RMB）。



读者可以使用如下的指令查询当前数据库支持的字符集。

```
SQL> SELECT userenv('language')
  2  FROM dual;

USERENV('LANGUAGE')
----------------------------------------------------
SIMPLIFIED CHINESE_CHINA.ZHS16GBK
```

其中 CHINESE_CHINA.ZHS16GBK 的组成是：语言_区域_字符集。显然这里的语言是中文，区域是中国，字符集是中文字符集。ZHS16GBK 的组成是：<语言><比特位><编码>，即语言是 ZHS，比特位是 16，编码方案是 GBK。字符集 ZHS16GBK 的含义是采用 GBK 编码的 16 位表示的简体中文。

在实例 3-11 中，已把表的第 2 列和第 3 列都改为本地货币格式，即修改了列的显示格式，如果该表有很多列做了类似的显示格式修改，则可以使用实例 3-12 和实例 3-13 所示的指令查看该列的显示格式设置。

实例 3-12　查看列 losal 的显示格式。

```
SQL> col losal
COLUMN   losal ON
FORMAT   $9999
```

实例 3-13　查看列 hisal 的显示格式。

```
SQL> col hisal
COLUMN   hisal ON
FORMAT   $9999
```

2. CLE(AR)

如果不需要某列的格式化设置，可以采用实例 3-14 所示的 col 指令进行删除，在这里删除列 losal 和 hisal 的格式化设置。

实例 3-14　删除列 losal 和 hisal 的格式化设置。

```
SQL> col losal clear
SQL> col hisal clear
```

为了验证是否删除列 losal 和 hisal 的格式化设置，可使用实例 3-15 查看这两列的格式化设置。

实例 3-15　查看这两列的格式化设置。

```
SQL> col losal
SP2-0046: COLUMN 'losal' 未定义
SQL> col hisal
SP2-0046: COLUMN 'hisal' 未定义
```

显然，在实例 3-14 中已删除列 losal 和 hisal 的格式化设置，并成功执行。

3. HEA[DING] text

在实例 3-11 中查询表 salgrade 时，列的属性如 losal、hisal 等显示不是很直观，毕竟不同国家

的人还是希望使用本国文字显示，而且这些列的属性名是程序员为了开发方便而随意命名的，这些名字并非对所有人而言都容易理解，所以在 SQL*Plus 中给出了一个使用 HEA[DING]（方括号表示省略，不必写出完整的单词）的格式化模式，如实例 3-16 所示。

实例 3-16　使用 HEADING 格式化列 losal 和 hisal。

```
SQL> col losal heading '低工资'
SQL> col hisal heading '高工资'
```

采用实例 3-17 验证上述的修改是否成功。

实例 3-17　验证实例 3-16 的格式化是否成功。

```
SQL> SELECT *
  2  FROM salgrade;

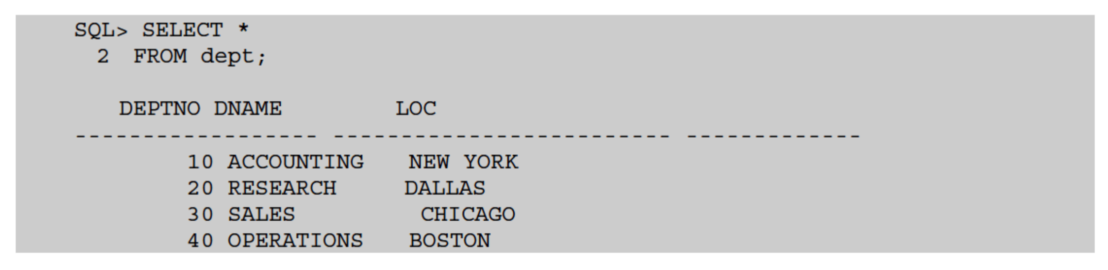
     GRADE            低工资            高工资
---------- --------------- ---------------
         1          RMB700         RMB1200
         2         RMB1201         RMB1400
         3         RMB1401         RMB2000
         4         RMB2001         RMB3000
         5         RMB3001         RMB9999
         1          RMB700         RMB1200
         2         RMB1201         RMB1400
         3         RMB1401         RMB2000
         4         RMB2001         RMB3000
         5         RMB3001         RMB9999
```

从实例 3-17 中可以看出，列 losal 和 hisal 都得到了格式化，输出显示为容易理解的“低工资”和“高工资”，想必这样的表格更容易让人接受吧！

4. JUS[TIFY] {L[EFT]|C[ENTER] |R[IGHT]}

JUS[TIFY] 的作用是调整列属性名字在当前显示字符宽度范围内的位置，可以在字符宽度范围内的左边、中间和右边放置。查询表 dept 数据，如实例 3-18 所示。

实例 3-18　查询表 dept 的全部数据。



```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
------------------ ------------------------ -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

从结果可以看出，列 DEPTNO 显示在输出字符宽度的右边，而列 DNAME 和 LOC 显示在输出字符宽度的左边，下面我们调整列 DNAME 和 LOC，使得他们显示在其字符宽度的中间，如实例 3-19 所示。

实例 3-19　格式化列 DNAME 和 LOC，使得列名显示在字符宽度的中间。

```
SQL> col dname jus center
```

```
SQL> col loc   jus center
```

利用实例 3-20 重新验证对列 DNAME 和 LOC 的格式化效果。

实例 3-20　验证对列 DNAME 和 LOC 的格式化效果。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO     DNAME           LOC
---------------- -------------------------- -----------------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH        DALLAS
        30 SALES            CHICAGO
        40 OPERATIONS      BOSTON
```

通过实例 3-19 的格式化设置，使得列 DNAME 和 LOC 的显示位置在其字符宽度内得到调整。当然读者也可以自行验证 jus 的两外两个参数：L、R。

5. NEWL[INE]

该格式化的作用是将从该列开始的所有列的显示另起一行，不过该格式化的用处不多，估计没有人愿意把数据显示得如此凌乱吧？我们只通过实例 3-21 让读者体会一下它的效果。

实例 3-21　使用 NEWL[INE]格式化列。

```
SQL> col dname newline
```

利用实例 3-22 验证上述代码格式化的效果。

实例 3-22　验证实例 3-21 的格式化效果。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO
----------
DNAME          LOC
-------------- -------------
        10
ACCOUNTING     NEW YORK
```

显然，newline 的格式化得到验证，即从列 DNAME 开始的所有列（DNAME、LOC）另起一行显示。在这种情况下，查看列中的数据不是很方便，只能沿着列名的位置向下寻找，显然让人很痛苦。在 SQL*Plus 中，如果显示窗口的宽度不够，则会自动另起一行显示，所以这个格式化指令用得很少，至少笔者没有用过。

6. NOPRI[NT]|PRI[NT]

或许从其英文含义中就可以理解该指令的作用，利用 NOPRI[NT]|PRI[NT]格式化后使得格式化的列数据不显示（NOPRI）或者显示（PRI），对表 dept 的 LOC 列实现 NOPRI 格式化，如实例 3-23 所示。

实例 3-23　使用 NOPRI[NT]|PRI[NT]格式化。

```
SQL> col loc noprint
```

利用实例 3-24 验证执行格式化的效果。

实例 3-24　验证格式化效果。

```
SQL> SELECT *
  2  FROM dept;

   DEPTNO DNAME
------------------- --------------
       10  ACCOUNTING
       20  RESEARCH
       30  SALES
       40  OPERATIONS
       50  MARKETING
       60  MARK

已选择 6 行。
```

此时，列 LOC 的数据没有显示，如果打算恢复该列的显示，可以使用如下命令：

```
SQL> col loc print
```

这里的格式化效果不再给出，请读者自行验证。当然如果想恢复该列的显示，也可以使用 CLE[AR]清除格式化，这样就清除了所有对该列的格式化设置，如实例 3-25 所示。

实例 3-25　清除对列 LOC 的所有格式化设置。

```
SQL> col loc clear
```

7. NUL[L] text

该格式化的目的是把列中值为空的字段用随后的 text 文本代替，为了测试 NUL[L] text 格式化效果，首先使用实例 3-26 向表 dept 中插入一行数据。

实例 3-26　向表 dept 中插入一行数据。

```
SQL> insert into dept (deptno,dname,loc)
  2           values(50,'Marcketing','');

已创建 1 行。
```

利用实例 3-27 验证是否插入数据。

实例 3-27　验证实例 3-26 是否成功插入数据。

```
SQL> SELECT *
  2  FROM dept;

   DEPTNO DNAME          LOC
----------------- -------------------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH        DALLAS
        30 SALES           CHICAGO
        40 OPERATIONS      BOSTON
        50 Marcketing
```

从输出结果可以看出，我们成功插入了一行数据，DEPTNO 为 50，DNAME 为 Marcketing，

LOC 为空值（没有显示），使用实例 3-28 将其格式化，希望在输出时将 LOC 为空值的字段显示为'TEMP'（表示临时）。

实例 3-28　格式化列 LOC，使得该列为空值的字段显示为'TEMP'。

```
SQL> col loc null 'TEMP'
```

利用实例 3-29 验证格式化效果。

实例 3-29　验证实例 3-28 的格式化效果。

```
SQL> SELECT *
  2  FROM dept;

   DEPTNO DNAME          LOC
----------------- -------------------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH        DALLAS
        30 SALES            CHICAGO
        40 OPERATIONS      BOSTON
        50 Marcketing         TEMP
```

从显示结果可以看出，第 5 行记录中列 LOC 为空值的字段填充为'TEMP'。需要注意的是：这里只是显示给用户的数据，而表中实际的数据没有任何变化。

8. ON|OFF

使用 OFF 格式化指令，将使之前列的所有格式化操作自动取消，以后任何的格式化该列的操作，虽然 SQL*Plus 不会报错，但是这些格式化修改都将无效。我们在实例 3-29 之后继续操作，对表 dept 的 LOC 列实现 OFF 格式化，如实例 3-30 所示。

实例 3-30　使用 OFF 格式化列 LOC。

```
SQL> col loc off
```

再次验证格式化效果：

```
SQL> SELECT *
  2  FROM dept;

   DEPTNO DNAME          LOC
------------------ -------------------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH        DALLAS
        30 SALES            CHICAGO
        40 OPERATIONS      BOSTON
        50 Marcketing
```

从结果可以看出，以前对列 LOC 的 NULL 格式化不再生效，下面我们重新对列 LOC 进行 NULL 格式化，并验证该格式化是否成功，如实例 3-31 所示。

实例 3-31　重新对列 LOC 进行 NULL 格式化。

```
SQL> col loc null 'temp'
SQL> SELECT *
```

```
  2  FROM dept;

   DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 Marcketing
```

可以看出在格式化时，系统没有报错，但是执行查询语句时，列 LOC 的 NULL 格式化没有生效。如果想继续在列 LOC 实现格式化操作，可使用 ON 格式化。

设置继续在列 LOC 实现格式化操作的代码如下。

```
SQL> col loc on
```

3.2.3 run 或 “/” 指令

在使用 SQL*Plus 操纵 SQL 语句时，往往会重复执行 SQL 缓冲区中的语句，我们先执行一个查询，如实例 3-32 所示。

实例 3-32　查询表 EMP 中的员工数据。

```
SQL> SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3  WHERE job = 'MANAGER';

EMPNO  ENAME     JOB        MGR HIREDATE         SAL
-------- -------------- -------------- ------ -------------- -------
   7566  JONES     MANAGER    7839 02-4 月 -81        2975
   7698  BLAKE     MANAGER    7839 01-5 月 -81        2850
   7782  CLARK     MANAGER    7839 09-6 月 -81        2450
```

如果期间关于'MANAGER'的记录被修改，或者因其他原因需要继续执行该指令，则需要使用 run 或者“/”指令。下面通过实例 3-33 验证该指令，请注意二者的区别。

实例 3-33　在实例 3-32 的查询后直接使用 run 和“/”指令。

```
SQL> run
  1  SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3* WHERE job = 'MANAGER'

EMPNO  ENAME     JOB        MGR HIREDATE         SAL
-------- -------------- -------------- ------ -------------------- -------
   7566  JONES     MANAGER    7839 02-4 月 -81        2975
   7698  BLAKE     MANAGER    7839 01-5 月 -81        2850
   7782  CLARK     MANAGER    7839 09-6 月 -81        2450

SQL> /

EMPNO  ENAME     JOB        MGR HIREDATE         SAL
------ ------------ ------------- ------ ---------------------- -------
   7566  JONES     MANAGER    7839 02-4 月 -81        2975
   7698  BLAKE     MANAGER    7839 01-5 月 -81        2850
```

```
    7782  CLARK       MANAGER     7839 09-6 月 -81          2450
```

显然，在输入 run 的时刻，SQL 语句有反馈被显示出来，而“/”则没有 SQL 语句的反馈。其实使用 run 也可以不显示 SQL 语句的反馈信息，需要设置 SQL*Plus 的环境变量 FEEDBACK。

3.2.4　L(list)指令和 n 指令

L(list)指令列出了当前 SQL 缓冲区中的 SQL 指令，在执行完实例 3-33 后，继续执行 L(list)指令，如实例 3-34 所示。

实例 3-34　L(list)指令列出当前 SQL 缓冲区中的 SQL 指令语句。

```
SQL> list
  1  SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3* WHERE job = 'MANAGER'未选定行

SQL>
```

注意此时在第 3 行有一个“*”号，因为 SQL*Plus 允许我们修改 SQL 缓冲区中的 SQL 语句，例如更改第 1 行，不需要查询 sal 列的信息等。“*”号就定位了要修改的行 ，实例 3-35 演示了如何定位要修改的行。

实例 3-35　定位要修改的行。

```
SQL> 1
  1* SELECT empno ,ename,job,mgr,hiredate,sal
SQL>
```

一旦输入行号 1，则提示定位在第一行。这样就为下节介绍的 change 指令做好了准备。

3.2.5　change 指令和 n (next)指令

执行完实例 3-35 后继续执行 change 指令，该指令用于修改某行的字段，如实例 3-36 所示。

实例 3-36　修改第 1 行的字段。

```
SQL> ch /sal/deptno
  1* SELECT empno ,ename,job,mgr,hiredate,deptno
SQL> /

EMPNO ENAME      JOB        MGR  HIREDATE      DEPTNO
----- ---------- --------- ----- ---------- ----------
 7566 JONES      MANAGER    7839 02-4 月 -81        20
 7698 BLAKE      MANAGER    7839 01-5 月 -81        30
 7782 CLARK      MANAGER    7839 09-6 月 -81        10
```

使用 change 指令把列 sal 改为列 deptno，一旦修改成功，则随后显示修改结果，可使用“/”指令验证是否修改成功。

n 指令用来修改整行的 SQL 语句，n 为 SQL 语句的行号，可利用随后的输入语句替换 n 行的 SQL 语句。

在实例 3-36 执行成功后，执行实例 3-37 来查看结果。

实例 3-37　通过 list 指令查看实例 3-36 的修改结果。

```
SQL> list
  1  SELECT empno ,ename,job,mgr,hiredate,deptno
  2  FROM emp
  3* WHERE job = 'MANAGER'
SQL> 1 SELECT empno,ename,job,mgr
SQL> list
  1  SELECT empno,ename,job,mgr
  2  FROM emp
  3* WHERE job = 'MANAGER'
SQL>
```

首先输入 list 指令，用于查询当前 SQL 缓冲区中的语句，输入"1 SELECT empno,ename,job,mgr"表示把第一行的 SQL 语句替换为"SELECT empno,ename,job,mgr"，执行后输入 list 再次验证，发现修改成功。

3.2.6　附加（a）指令

附加（a）指令就是在某行末尾添加一些语句或属性信息，首先利用实例 3-38 来查询 demp 表中所有部门的名字。

实例 3-38　查询 demp 表中所有部门的名字。

```
SQL> SELECT dname
  2  FROM dept;

DNAME
--------------
ACCOUNTING
RESEARCH
SALES
OPERATIONS
Marcketing
```

如果想同时知道每个部门的所在地，可以使用如下方式修改 SQL 缓冲区中的语句。首先用 list 指令查询当前的指令，如实例 3-39 所示。

实例 3-39　用 list 指令查询当前的指令。

```
SQL> list
  1  SELECT dname
  2* FROM dept
```

显然，此时"*"号在第 2 行，而不是我们要修改的第 1 行。

若要定位要修改的行，代码如下所示：

```
SQL> 1
  1* SELECT dname
```

此时，"*"号定位在第 1 行，说明已经将 SQL 缓冲区中的第 1 行语句定位为当前行。接着可以使用 a（附加）指令将"，loc"添加在第 1 行 SQL 语句的末尾，如实例 3-40 所示。

实例 3-40　使用 a（附加）指令将“，loc”添加在第 1 行 SQL 语句的末尾。

```
SQL> a ,loc
  1* SELECT dname,loc
```

再次使用 list 指令验证修改效果。

```
SQL> list
  1  SELECT dname,loc
  2* FROM dept
```

显然，修改已成功，在 SQL 语句的第 1 行添加了列 LOC，此时可以使用 run 命令继续执行，以输出希望的数据，如实例 3-41 所示。

实例 3-41　使用 run 指令查看修改结果。

```
SQL> run
  1  SELECT dname,loc
  2* FROM dept
DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON
Marcketing
```

3.2.7　del 指令

del 指令用于删除 SQL 缓冲区中的某行 SQL 语句。其语法格式是：del n（n 为行号），所以在删除某行之前需要做一些工作：首先需要输入 list 指令以验证当前 SQL 缓冲区中的 SQL 语句，在确定要删除的行后执行 del n 指令，再次输入 list 指令验证删除效果。

为了验证整个过程，可执行 SQL 查询，如实例 3-42 所示。

实例 3-42　执行 SQL 查询。

```
SQL> SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3  WHERE job = 'MANAGER'
  4  order by sal;
```

输入 list 指令以验证当前 SQL 缓冲区中的 SQL 语句，如实例 3-43 所示。

实例 3-43　使用 list 指令验证当前 SQL 缓冲区中的 SQL 语句。

```
SQL> l
  1  SELECT empno ,ename,job,mgr,hiredate,sal
  2  FROM emp
  3  WHERE job = 'MANAGER'
  4* order by sal
```

若要删除第 3 行和第 4 行，则代码如实例 3-44、实例 3-45 所示。

实例 3-44　删除当前 SQL 缓冲区中的 SQL 语句的行。

```
SQL> del 4
```

```
SQL> del 3
```

实例 3-45　使用 list 指令查询实例 3-44 删除后的 SQL 指令。

```
SQL> list
  1  SELECT empno ,ename,job,mgr,hiredate,sal
  2* FROM emp
```

再通过实例 3-46 验证修改后的结果。

实例 3-46　验证实例 3-44 的删除效果。

```
SQL> /

EMPNO ENAME      JOB         MGR HIREDATE         SAL
------- ------------- ----------- --------- -------------- --------------
  7369 SMITH      CLERK      7902 17-12 月-80         800
  7499 ALLEN      SALESMAN   7698 20-2 月 -81        1600
  7521 WARD       SALESMAN   7698 22-2 月 -81        1250
  7566 JONES      MANAGER    7839 02-4 月 -81        2975
  7654 MARTIN     SALESMAN   7698 28-9 月 -81        1250
  7698 BLAKE      MANAGER    7839 01-5 月 -81        2850
  7782 CLARK      MANAGER    7839 09-6 月 -81        2450
  7839 KING       PRESIDENT       17-11 月-81        5000
  7844 TURNER     SALESMAN   7698 08-9 月 -81        1500
  7900 JAMES      CLERK      7698 03-12 月-81         950
  7902 FORD       ANALYST    7566 03-12 月-81        3000

EMPNO ENAME      JOB         MGR HIREDATE         SAL
------ --------------- ------------ ----- ------------ -------------
  7934 MILLER     CLERK       7782 23-1 月 -82           1300

已选择 12 行。
```

3.2.8　set line 指令

该指令的使用格式为：set line {80/n}，作用是将查询的数据输出设置为 n 个字符宽度显示。默认为以 80 个字符的宽度输出。如果一个输出由于显示宽度不够使得有的列错行，这样的数据输出就很难让人接受。下面查询 scott 用户下的 EMP 表的全部信息，代码如实例 3-47 所示。

实例 3-47　查询表 EMP 的全部数据。

```
SQL> SELECT *
  2  FROM emp;

    EMPNO ENAME      JOB        MGR HIREDATE       SAL       COMM
----------- -------------- --------- ------- --------- ---------- ----------
   DEPTNO
------------------
     7369 SMITH      CLERK       7902 17-12 月-80      800
       20

     7499 ALLEN    SALESMAN     7698 20-2 月 -81      1600        300
       30
..............（省略了部分数据）
```

此时，采用默认的 80 个字符宽度，显然这样的宽度是不够的。使用 set line n 指令设置输出的显示字符宽度为 100 个字符，以改善数据的显示效果，如实例 3-48 所示。

实例 3-48　使用 set line 格式化行的长度并继续查询。

```
SQL> set line 100
SQL> /

EMPNO ENAME      JOB      MGR HIREDATE     SAL     COMM     DEPTNO
----- ------ ----------- ------ ------------ ------ -------- ------
  7369 SMITH    CLERK      7902 17-12 月-80     800                   20
  7499 ALLEN   SALESMAN  7698 20-2 月 -81     1600       300           30
  7521 WARD    SALESMAN  7698 22-2 月 -81     1250       500           30
  7566 JONES    MANAGER   7839 02-4 月 -81     2975                   20
  7654 MARTIN  SALESMAN   7698 28-9 月 -81     1250      1400          30
..............（省略了部分数据）
```

使用 set line 100 的指令将显示的行的长度设置为 100 个字符，再次查询表 EMP 中的全部数据时，就不会出现如实例 3-47 所示的拐行现象了。

3.2.9　spool 指令

spool 指令的作用是把用户输入的 SQL 语句和查询结果存储到指定的文件中，下面通过实例 3-49 来介绍如何使用 spool 指令。

实例 3-49　查看 spool 的状态。

```
SQL> show spool
spool OFF
```

这里使用 SHOW 查看 SQL*Plus 的参数 spool 的状态，该指令为关闭状态，那么如何开启并使用 spool 功能呢？实例 3-50 说明了用法。

实例 3-50　启动 spool 并将查询结果存储到.txt 文件中。

```
SQL> spool d:\spool_test
SQL> show spool
spool ON
```

这个实例的作用是开启 spool 并把接下来用户输入的 SQL 语句和查询结果存储到指定的 d:\spool_test 文件中。可以看到参数 spool 的状态为 ON。执行实例 3-51、实例 3-52。

实例 3-51　执行查询语句。

```
SQL>  SELECT empno, ename, job, mgr, sal
  2   FROM emp
  3   WHERE job = 'MANAGER';

EMPNO ENAME      JOB         MGR     SAL
---------- ----------------- ------------------- ----------- -----------
  7566 JONES      MANAGER    7839      2975
  7698 BLAKE      MANAGER    7839      2850
 7782 CLARK      MANAGER    7839      2450
```

实例 3-52　关闭 spool 功能。

```
SQL> spool off
```

执行完实例 3-51 和实例 3-52 后，在 D 盘根目录下生成一个 spool_test.lst 文件，用记事本打开该文件，内容如图 3-10 所示。

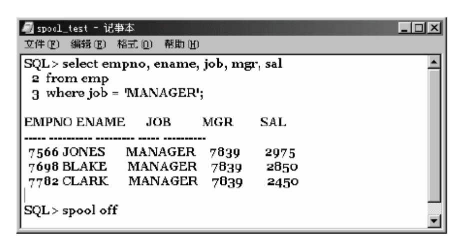


图 3-10　spool_test 文件内容



当用户查询大输出量的数据时，为了方便可以使用该指令，这样既可以保存输出记录，又方便使用记事本的工具查找相应的数据。

3.2.10　脚本文件

脚本文件就是由一系列 SQL 语句和 SQL*Plus 指令组成的，下面通过实例来演示如何生成脚本文件。先执行一个查询语句，如实例 3-53 所示。

实例 3-53　查询表 EMP 中部分员工的信息。

```
SQL> SELECT empno, ename, job, mgr, hiredate, sal
  2  FROM emp
  3  WHERE job = 'MANAGER'
  4  order by sal;

EMPNO ENAME      JOB          MGR HIREDATE       SAL
----- ---------- ----------- ---------------- -------------- -------
  7782 CLARK      MANAGER     7839 09-6 月 -81       2450
  7698 BLAKE      MANAGER     7839 01-5 月 -81       2850
  7566 JONES      MANAGER      7839 02-4 月 -81       2975
```

此时在 SQL 缓冲区中存储了上述 SQL 语句，接着执行实例 3-54，将上述 SQL 语句保存在指定目录下的文件中，注意存储目录必须存在，即此时 SQL*Plus 并不负责建立目录，只负责建立脚本文件。

实例 3-54　创建脚本文件。

```
SQL> save d:\SELECT_emp
已创建文件 d:\SELECT_emp.sql
```

显然，系统提示已经创建了脚本文件，文件类型为.sql，在目录“d:\”下可以找到刚才创建的 SELECT_emp.sql 文件。

在实例 3-54 中创建了 SELECT_emp.sql 脚本文件后，应该如何使用该脚本文件呢？下面使用实例 3-55 和实例 3-56 来进行说明。

实例 3-55　运行脚本文件。

```
SQL> @d:\SELECT_emp

EMPNO ENAME      JOB          MGR HIREDATE       SAL
------ --------------- --------------- -------- ----------------- -------
  7782 CLARK      MANAGER     7839 09-6 月 -81       2450
  7698 BLAKE      MANAGER     7839 01-5 月 -81       2850
  7566 JONES      MANAGER      7839 02-4 月 -81       2975
```

实例 3-56　使用 start 运行脚本文件。

```
SQL> start d:\SELECT_emp

EMPNO ENAME      JOB          MGR HIREDATE       SAL
------ ------------- ------------- ------- ----------------- --------
  7782 CLARK      MANAGER     7839 09-6 月 -81       2450
  7698 BLAKE      MANAGER     7839 01-5 月 -81       2850
  7566 JONES      MANAGER      7839 02-4 月 -81       2975
```

通过上述两个实例可以说明如何运行脚本文件，即使用@或者 start 指令启动脚本文件，把文件中的 SQL 指令装入 SQL 缓冲区。在实际维护中，如果用户需要频繁地使用某些 SQL 指令，就可以编写脚本文件，从而提高工作效率。

在创建了脚本文件后，如果需要修改文件内容应该怎么办呢？或许读者已经知道了，可以使用记事本打开脚本文件，这样做是可行的。但是，如果运行在 SQL*Plus 模式下，应该怎样处理呢？下面通过实例 3-57 演示如何编辑脚本文件。

实例 3-57　使用 get 指令将脚本文件装入 SQL 缓冲区。

```
SQL> get d:\SELECT_emp
  1  SELECT empno, ename, job, mgr, hiredate, sal
  2  FROM emp
  3  WHERE job = 'MANAGER'
  4* order by sal
```

在使用 get 指令后，即可将脚本文件中的 SQL 语句装入 SQL 缓冲区，这样就可以使用 SQL*Plus 指令进行修改了。

SQL*Plus 也提供了一个 edit 指令，可通过调用操作系统编辑软件来直接修改该文件。编辑脚本文件的代码如下。

```
SQL> edit d:\SELECT_emp
```

在输入上述指令后，可以直接打开该文件，如图 3-11 所示，直到关闭该文件，SQL*Plus 才恢复到“SQL>”状态。

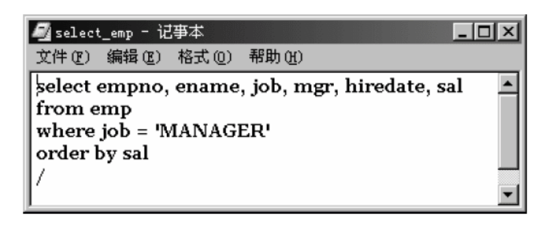


图 3-11　SELECT_emp.sql 文件内容

3.3　SQL*Plus的环境变量

3.3.1　ECHO 环境变量

SQL*Plus 的 ECHO 环境变量有两种状态 ON 和 OFF。如果 ECHO 环境变量的状态为打开，则使用脚本文件执行 SQL 语句时，脚本文件中的 SQL 语句会输出显示，否则，不显示脚本文件中的 SQL 语句。

首先查看该变量的状态，如实例 3-58 所示。

实例 3-58　查询 ECHO 状态。

```
SQL> show echo
echo OFF
```

此时，ECHO 是关闭的，所以不会显示脚本文件中的 SQL 语句。测试结果如实例 3-59 所示。

实例 3-59　在 ECHO 关闭状态下查询脚本文件。

```
SQL> @d:\SELECT_emp

    EMPNO  ENAME      JOB          MGR  HIREDATE       SAL
------------ ------- -------------- ----------- --------- --------------
     7782  CLARK      MANAGER       7839  09-6 月 -81       2450
     7698  BLAKE      MANAGER       7839  01-5 月 -81       2850
     7566  JONES      MANAGER        7839  02-4 月 -81       2975
```

此时，将 ECHO 环境变量的状态设置为 ON，如实例 3-60 所示。

实例 3-60　将 ECHO 环境变量的状态设置为 ON。

```
SQL> set echo on
```

使用实例 3-61 查看 ECHO 环境变量的当前状态。

实例 3-61　查看 ECHO 环境变量的当前状态。

```
SQL> show echo
echo ON
```

现在使用实例 3-62 测试执行脚本文件。

实例 3-62　在 ECHO 打开状态下查询脚本文件。

```
SQL> @d:\SELECT_emp
SQL> SELECT empno, ename, job, mgr, hiredate, sal
  2  FROM emp
  3  WHERE job = 'MANAGER'
  4  order by sal
  5  /

    EMPNO  ENAME      JOB          MGR  HIREDATE       SAL
------------------ ----------- ------------- -------------- --------------
     7782  CLARK      MANAGER       7839  09-6 月 -81       2450
     7698  BLAKE      MANAGER       7839  01-5 月 -81       2850
```

```
        7566 JONES       MANAGER       7839 02-4 月 -81       2975
```

测试结果表明，在 ECHO 环境变量的当前状态为 ON 时执行脚本文件，则可显示脚本文件中的 SQL 指令。

3.3.2　FEEDBACK 环境变量

FEEDBACK 环境变量用于控制查询输出的数据行数是否显示，以及记录数达到什么值时才显示数据行数。其语法格式为：SET FEED[BACK]{6/n/on/off}。

首先，使用实例 3-63 查看 FEEDBACK 变量的当前值。

实例 3-63　查看 FEEDBACK 变量的当前值。

```
SQL> show feedback
```

以上代码用于 6 行或更多行的 FEEDBACK ON 说明，但数据行数等于或多于 6 行时，才显示数据行数，实例 3-64 用于查询表 salgrade 的数据。

实例 3-64　查询表 salgrade 的数据。

```
SQL> SELECT *
  2  FROM salgrade;

    GRADE      LOSAL      HISAL
---------------- --------------------- -----------------
        1        700       1200
        2       1201       1400
        3       1401       2000
        4       2001       3000
        5       3001       9999
        1        700       1200
        2       1201       1400
        3       1401       2000
        4       2001       3000
        5       3001       9999

已选择 10 行。
```

显然，因为输出数据行数大于 6，所以显示计算得到的数据行数，也可以更改变量 FEEDBACK 的参数，如实例 3-65 所示。

实例 3-65　更改变量 FEEDBACK 的参数。

```
SQL> set feedback 12
```

为了验证修改是否成功，可使用实例 3-66 查询当前 FEEDBACK 变量的状态。

实例 3-66　查询当前 FEEDBACK 变量的状态。

```
SQL> show feedback
```

以上代码用于 12 行或更多行的 FEEDBACK ON，显然修改成功，此时为了测试修改结果，可再次执行实例 3-64，结果如下：

```
    GRADE      LOSAL      HISAL
```

```
----------------- ------------------- -------------------
        1        700       1200
        2       1201       1400
        3       1401       2000
        4       2001       3000
        5       3001       9999
        1        700       1200
        2       1201       1400
        3       1401       2000
        4       2001       3000
        5       3001       9999
```

因为当前的 FEEDBACK 参数为 12，所以小于 12 行的数据行数的提示是不显示的。

3.4 本章小结

本章主要讲解了 SQL*Plus 工具，该工具是 Oracle 提供的用户操作数据库的人机接口，通过这个接口用户在拥有一定权利的条件下可以操纵数据库、更改数据库的各种系统配置。SQL*Plus 工具作为一个应用工具软件，提供了一系列的编辑指令，使得用户可以更方便地操作和使用 SQL*Plus 工具。如果用户经常使用同样的 SQL 语句，为了提高输入效率可以使用 SQL*Plus 提供的脚本文件。为了方便用户数据和其他信息的显示，SQL*Plus 工具还提供了环境变量。

通过本章的学习，读者应该能够熟练使用 SQL*Plus 的各种编辑指令和格式化指令，并掌握如何编辑和运行脚本文件，这些都是日常维护中必备的技能。

第 4 章

SQL语言概述

SQL 语言是“结构化查询语言”的意思，即 Structured Query Language。SQL 语言涉及的语法简单，语义明了，如果读者稍微懂些英文，则可很容易掌握 SQL 语言。通过 SQL 语言可以检索和维护数据库，编写涉及数据库操作的应用程序或脚本语言。在使用这些 SQL 语句时，可以使用一些函数来处理输出结果或者通过分组函数使得输出数据更加友好。

4.1 SQL语句的分类

SQL 语句按照其功能分为 5 类，即数据查询语句、数据操纵语句、数据定义语句、事务控制语句和数据控制语句。下面通过 SQL 语句的关键字依次简单介绍这些语句的功能，在本书后续的章节中读者可以体会如何使用这些语句，以及使用这些语句的场合。

1. 数据查询语句

SELECT 语句的功能是从数据库中获得用户数据，如查询一个表中的全部数据等。

2. 数据操纵语句（DML）

- INSERT：该语句的功能是向表中添加记录。
- UPDATE：该语句的功能是更新表中的数据，通常和 WHERE 条件语句一起使用。
- DELETE：删除表中的数据。

3. 数据定义语句（DDL）

- CREATE：创建数据库对象，如表、索引、视图等。
- ALTER：改变系统参数，如改变 SGA 的大小等。
- DROP：删除一个对象，如删除一个表、索引或者序列号等。
- RENAME：重命名一个对象。
- TRUNCATE：截断一个表。

4. 事务控制语句

- COMMIT：用于提交由 DML 语句操作的事务。
- ROLLBACK：用于回滚 DML 语句改变了的数据。

5．数据控制语句

- GRANT：用于授予用户访问某对象的特权。
- REVOKE：用于回收用户访问某对象的特权。

4.2　数据查询语句

Oracle 的数据查询语句，即 SELECT 语句。如果需要检索数据库中的数据，就需要使用该语句。在使用 SELECT 语句时，必须有相应的 FROM 子句。当需要复杂查询时可以使用 WHERE 子句。把整个查询语句中的 SELECT、FROM 和 WHERE 称为关键字，下面将详细介绍查询语句的用法和各个关键字的含义。

4.2.1　语法及书写要求

一个简单的 SELECT 语句至少包含一个 SELECT 子句和一个 FROM 子句。其中 SELECT 子句用于指明要显示的列，而 FROM 子句用于指明需要查询的表，该表包含了在 SELECT 子句中的列。其语法格式如下。

```
SELECT *|{[DISTINCT] column | expression [alias],……}
FROM    table;
```



在上述语法规则中，“|”号表示或的关系，“[]”表示可选。

对上述代码的说明如下。

- SELECT：选择一个列或多个列。
- *：选择表中所有的列。
- DISTINCT：去掉列中重复的值。
- column|expression：选择列的名字或表达式。
- alias：为指定的列设置不同的标题。
- FROM table：指定要选择的列所在的表，即对哪个表进行数据检索。

其中有几个术语需要读者分辨清楚，因为在接下来的内容中将多次用到，它们是关键字、子句和语句，其区别如下。

- 关键字：它是一个单独的 SQL 元素，如 SELECT、FROM 等都是关键字，并且要求关键字不能简写，如写成 SEL、FRO 等都是不允许的，但是不要求必须大写，这里采用大写是 Oracle 推荐的写法，即关键字都大写而其他小写以示区分。
- 子句：子句是一个 SQL 语句的一部分，它不是一个可执行的 SQL 语句，如“SELECT *”就是一个子句。

- 语句：语句由一个或多个子句组成，它是可执行的，例如“SELECT * FROM dept”就是一个语句。在书写语句时，读者最好采取每个子句一行的习惯，这样可增强可读性。

4.2.2　查询表中的全部数据

先使用一个实例说明如何实现一个简单的查询，此时我们使用用户 scott（该用户在创建数据库时会自动创建）的 dept 表，如实例 4-1 所示。

实例 4-1　查询 scott 用户的 dept 表的全部内容。

```
SQL> conn scott/tiger
已连接。
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

首先使用 scott 用户登录，该用户的默认密码是 tiger，此时不区分用户名和密码的大小写。然后输入一个查询语句，该语句的作用是查询表 dept 中的所有列的数据。

这里“*”号的含义是选择表中的所有列，FROM 关键字后是表名。表 dept 是一个部门表，该表有三列，分别是 DEPTNO（部门号）、DNAME（部门名称）和 LOC（部门所在地）。

还有一种查询方式可以实现查询表中的所有列的数据，即在 SELECT 关键字后输入所有列的名字，名字之间用逗号分开。如实例 4-2 中，我们重新查询表 dept 中的所有列的数据。

实例 4-2　重新查询表 dept 的全部内容。

```
SQL> SELECT deptno,dname,loc
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

在实例 4-2 中，SELECT 关键字之后的列名全部利用逗号分开。



实例 4-1 和实例 4-2 都在查询用户 scott 的表，如果想要操作顺利，需要读者使用 scott 用户登录，如果使用 system 用户登录，就会提示出现错误，如实例 4-3 所示。

实例 4-3　使用 system 用户登录，该用户的默认密码是 manager。

```
SQL> conn system/manager
已连接。
SQL> SELECT *
  2  FROM dept;
```

```
FROM dept
     *
ERROR 位于第 2 行:
ORA-00942: 表或视图不存在
```

此时如果将 FROM 子句改为 FROM scott.dept，该语句就会顺利执行，如实例 4-4 所示。

实例 4-4　在 SYSTEM 用户模式下使用“模式名.表名”的方式查询表数据。

```
SQL> conn system/manager
已连接。
SQL> SELECT *
  2  FROM scott.dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

因为 system 为系统管理员用户，所以他有权限操作 scott 用户的对象，使用 system 用户登录，在查询用户 scott 的对象时只须在对象前指明是该用户的对象就可以了。

4.2.3　查询特定的列

在实际应用中，并不是表中所有的列都需要查询，用户只需要查询所需要的列，即在 SELECT 关键字后输入要查询的列名，如实例 4-5 所示。

实例 4-5　查询表 dept 的特定列的数据。

```
SQL> SELECT dname,loc
  2  FROM dept;

DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON
```

其实，在 SELECT 之后可以输入表中存在的任意列，并且列的顺序没有要求，数据的显示将以用户输入的列的顺序为基准，如实例 4-6 所示。

实例 4-6　查询表 dept 中的任意列的数据。

```
SQL> SELECT dname,loc,deptno
  2  FROM dept;

DNAME          LOC               DEPTNO
-------------- ------------- ----------
ACCOUNTING     NEW YORK              10
RESEARCH       DALLAS                20
SALES          CHICAGO               30
OPERATIONS     BOSTON                40
```

4.2.4　查询特定条件的表

如果用户想查询一个特定条件的表该怎么办呢？Oracle 提供了 WHERE 子句来限制查询条件，WHERE 子句可以限制选择的行数，这样就可以实现满足一定条件的数据查询，例如实例 4-7 用于查询 scott 用户的表 dept 中满足部门所在地为 CHICAGO 的部门所有信息。

实例 4-7　使用 WHERE 子句查询 scott 用户的表 dept 的全部数据。

```
SQL> SELECT *
  2  FROM dept
  3  WHERE loc = 'CHICAGO';

    DEPTNO DNAME          LOC
----------------- --------------------- -------------
        30  SALES          CHICAGO
```

上述查询虽然使用了 SELECT *子句，但是 WHERE 子句限制了查询的结果必须是 loc 为 CHICAGO 的部门信息，所以限制了查询的行数。当然，用户也可以输入其他限制性条件，如查询部门号小于 30 的部门信息，如实例 4-8 所示。

实例 4-8　查询表 dept 中部门号小于 30 的所有数据。

```
SQL> SELECT *
  2  FROM dept
  3  WHERE deptno < 30;

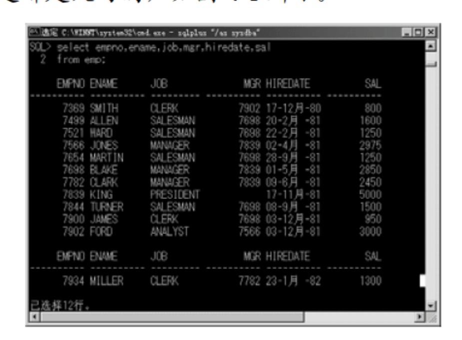
    DEPTNO DNAME          LOC
---------- ------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
```

WHERE 子句中的条件可以根据需要，通过各种算数或逻辑运算符实现条件限制。



在前面介绍的查询语句中，显示的结果都是 Oracle 提供的，我们并没有对显示的信息做任何的修改，即数据的显示结果是 Oracle 的默认结果。在 Oracle 中列标题的显示满足如下规则：

- 字符和日期型的列标题靠在显示宽度的左边。
- 数字型的列标题靠在显示宽度的右边。
- 默认的列标题都是大写的，如图 4-1 所示。



```
SQL> select empno,ename,job,mgr,hiredate,sal
  2  from emp;

     EMPNO ENAME      JOB              MGR HIREDATE          SAL
---------- ---------- --------- ---------- ---------- ----------
      7369 SMITH      CLERK           7902 17-12月-80        800
      7499 ALLEN      SALESMAN        7698 20-2月 -81       1600
      7521 WARD       SALESMAN        7698 22-2月 -81       1250
      7566 JONES      MANAGER         7839 02-4月 -81       2975
      7654 MARTIN     SALESMAN        7698 28-9月 -81       1250
      7698 BLAKE      MANAGER         7839 01-5月 -81       2850
      7782 CLARK      MANAGER         7839 09-6月 -81       2450
      7839 KING       PRESIDENT            17-11月-81       5000
      7844 TURNER     SALESMAN        7698 08-9月 -81       1500
      7900 JAMES      CLERK           7698 03-12月-81        950
      7902 FORD       ANALYST         7566 03-12月-81       3000

     EMPNO ENAME      JOB              MGR HIREDATE          SAL
---------- ---------- --------- ---------- ---------- ----------
      7934 MILLER     CLERK           7782 23-1月 -82       1300

已选择12行。
```

图 4-1　列标题的默认属性

4.2.5　在查询中使用别名

在使用 SELECT 语句时，SQL*Plus 使用选择的列名作为列标题，并且采用大写方式显示。但由于表中的列名是数据库开发人员或程序员为了编程的需要而设计的，这样的列标题可能因不具备描述性而难以理解，因此 Oracle 提供了列别名更改列标题的显示方式，如实例 4-9 所示。

实例 4-9　通过别名更改列标题的查询。

```
SQL>SELECT empno,ename employee_name,sal AS salary,deptno "Deptmentnumber"
  2* FROM emp

  EMPNO EMPLOYEE_NAME           SALARY Deptmentnumber
-------------- ------------------------ ---------- ----------------
     7369 SMITH                    800           20
     7499 ALLEN                   1600           30
     7521 WARD                    1250           30
     7566 JONES                    2975           20
     7654 MARTIN                  1250           30
     7698 BLAKE                    2850           30
     7782 CLARK                   2450           10
     7839 KING                     5000           10
     7844 TURNER                  1500           30
     7900 JAMES                     950           30
     7902 FORD                     3000           20

   EMPNO EMPLOYEE_NAME           SALARY Deptmentnumber
-------------- ------------------------ ---------- --------------------
     7934 MILLER                  1300           10

已选择 12 行。
```

创建别名时，应在列名后使用 AS 关键字，之后紧跟别名或者在列名后加空格，之后为别名，如上例中的 ename employee_name 和 sal AS salary，但是此时的列标题在显示时为别名的大写格式。如果要保持别名的格式，可以使用双引号，例如此时的列标题 deptno "Deptmentnumber"。

4.2.6　在查询中使用算数运算符

算数运算符，即加、减、乘、除 4 种运算："+"、"-"、"*"、"/"，可使用算数运算符实现对日期型和数字型列的算数操作，创建一个具有算数运算的表达式，丰富查询的显示结果。

例如在 scott 用户的 EMP 表中，可使用实例 4-10 查询员工的名字和年薪。

实例 4-10　查询 scott 用户的 EMP 表中员工的名字和年薪。

```
SQL> SELECT ename "员工姓名",sal*12 "年薪"
  2  FROM emp
  3  WHERE job = 'MANAGER';

员工姓名             年薪
--------------------- ----------
JONES              35700
BLAKE              34200
CLARK              29400
```

其他运算符的使用规则类似，读者可以自行测试，如为所有 job= "SALESMAN" 的员工月薪增加 500。

算数运算符遵循一定的优先顺序，即“乘除”优先于“加减”，而“乘除”具有同等优先权，“加减”也具有同等优先权。同等优先权的运算符按照从左到右的顺序计算。

如“sal*12 + 1000”，先计算 sal*12，再加 1000 就是该表达式的最后计算结果，如实例 4-11 所示。

实例 4-11　在查询中使用算数运算符。

```
SQL>SELECT ename "员工姓名",sal*12+1000 "年薪"
  2  FROM emp
  3* WHERE job = 'MANAGER'

员工姓名          年薪
---------- ----------
JONES           36700
BLAKE           35200
CLARK           30400
```

4.2.7　在查询中使用 DISTINCT 运算符

DISTINCT 运算符可使查询的结果没有重复内容，如需要查询 scott 用户的 EMP 表中有多少个 JOB。我们先用实例 4-12 测试不使用 DISTINCT 的查询结果，再使用实例 4-13 测试使用 DISTINCT 的查询结果，通过两个结果的对比，读者可以清晰地体会到使用 DISTINCT 的区别。

实例 4-12　查询表 EMP 中的 JOB 名。

```
SQL> SELECT job
  2  FROM emp;

JOB
---------
CLERK
SALESMAN
SALESMAN
MANAGER
SALESMAN
MANAGER
MANAGER
PRESIDENT
SALESMAN
CLERK
ANALYST

JOB
---------
CLERK

已选择 12 行。
```

实例 4-12 的选择结果有 12 行，重复的 JOB 内容也显示在结果中，显然这样的结果不是我们想要的，而实例 4-13 通过使用 DISTINCT 关键字可实现不重复查询。

实例 4-13 使用 DISTINCT 关键字实现不重复查询表 emp 中的 JOB 名。

```
SQL> SELECT distinct job
  2  FROM emp;

JOB
---------
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

在 SELECT 关键字后紧跟 DISTINCT 关键字，使得选择的行没有重复的结果，但是如果 DISTINCT 关键字后有多个列，又该如何处理呢？如实例 4-14 所示。

实例 4-14 使用 DISTINCT 关键字实现多列查询。

```
SQL> SELECT distinct deptno,job
  2  FROM emp;

  DEPTNO JOB
--------------- ----------------
       10 CLERK
       10 MANAGER
       10 PRESIDENT
       20 ANALYST
       20 CLERK
       20 MANAGER
       30 CLERK
       30 MANAGER
       30 SALESMAN

已选择 9 行。
```

可以看到，此时使用 DISTINCT 关键字，结果中多个列的组合均没有重复的结果，即每一行的数据不完全相同。

4.2.8 在查询中使用连接运算符

连接运算符可以把列与其他列连接起来，也可以把列与字符串连接起来。连接符是两个竖线“||”，在连接字符串时使用单引号“'”。实例 4-15 是使用连接运算符的示例。

实例 4-15 使用连接运算符“||”。

```
SQL> SELECT ename  ||' is a '||job ||' and 1month salary is:'|| sal  As
  2  "The imployees's information" FROM emp;

The imployees's information
--------------------------------------------------------------------
SMITH is a CLERK and 1month salary is:800
ALLEN is a SALESMAN and 1month salary is:1600
WARD is a SALESMAN and 1month salary is:1250
JONES is a MANAGER and 1month salary is:2975
MARTIN is a SALESMAN and 1month salary is:1250
```

```
BLAKE is a MANAGER and 1month salary is:2850
CLARK is a MANAGER and 1month salary is:2450
KING is a PRESIDENT and 1month salary is:5000
TURNER is a SALESMAN and 1month salary is:1500
JAMES is a CLERK and 1month salary is:950
FORD is a ANALYST and 1month salary is:3000

The imployees's information
------------------------------------------------------------------
MILLER is a CLERK and 1month salary is:1300

已选择 12 行。
```

该实例中使用了 4 个连接运算符，即把三列（ename、job 和 sal）和两个字符串（'is a'和'and 1 month salary is: '）连接起来。显然这样的显示结果更容易阅读。在上例中，我们也使用了别名，即将显示的信息设置为一个列标题，即“The imployees's information”。

4.2.9　在查询中使用的书写规范

在书写 SQL 语句时，不区分大小写，如 SELECT 和 select 都是允许的。但是关键字不能跨行书写，也不能缩写，例如 SELECT 不能写成 SEL。一个 SQL 语句可以有多行。

Oracle 推荐了书写 SQL 语句的规范，使用该规范可使 SQL 语句更加容易阅读，并且容易区分 SQL 语句的关键字和其他对象名。推荐的规范如下。

- SQL 语句的关键字要大写，对象名小写。
- 缩进对齐，这样便于阅读。
- 每个子句一行。

下面给出实例 4-16。

实例 4-16　查询表 EMP 中工资大于 1500 的员工信息。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal
  2  FROM emp
  3  WHERE  sal > 1500;

    EMPNO ENAME      JOB              MGR HIREDATE          SAL
---------- ---------- --------- ---------- ---------- ----------
     7499 ALLEN      SALESMAN        7698 20-2 月 -81       1600
     7566 JONES      MANAGER         7839 02-4 月 -81       2975
     7698 BLAKE      MANAGER         7839 01-5 月 -81       2850
     7782 CLARK      MANAGER         7839 09-6 月 -81       2450
     7839 KING       PRESIDENT            17-11 月-81       5000
     7902 FORD       ANALYST         7566 03-12 月-81       3000

已选择 6 行。
```

每个子句一行使得阅读更加方便，通过 SQL 关键字大写，使得它们与 Oracle 对象区分开来，这样整段代码就很清晰了。

4.3 单行函数

为了方便数据库的操作，Oracle 提供了各种函数的操作，单行函数分为字符型单行函数、数字型单行函数和日期型单行函数。本节将依次讲解这三类函数。

4.3.1 字符型单行函数

字符型单行函数接收一个字符输入，并且返回一个计算结果，该结果可以是字符型，也可以是数字型。常用的单行字符型函数如下。

1. LOWER

其函数格式为：LOWER(column | expression)，函数功能是把字符串转换成小写，如实例 4-17 所示。

实例 4-17 使用单行函数 LOWER()。

```
SQL> SELECT LOWER('Structured Query Language')
  2  FROM dual;

LOWER('STRUCTUREDQUERYLAN
-------------------------
structured query language
```

2. UPPER

其函数格式为：UPPER(column | expression) ，函数功能是把字符串转换成大写，如实例 4-18 所示。

实例 4-18 使用单行函数 UPPER()。

```
SQL> SELECT UPPER('Structured Query Language')
  2  FROM dual;

UPPER('STRUCTUREDQUERYLAN
-------------------------
STRUCTURED QUERY LANGUAGE
```

3. INITCAP

其函数格式为：INITCAP(column | expression)，其功能是把字符串的首字母大写，如实例 4-19 所示。

实例 4-19 使用单行函数 INITCAP()。

```
SQL> SELECT INITCAP('structured query language')
  2  FROM dual;

INITCAP('STRUCTUREDQUERYL
-------------------------
Structured Query Language
```

4. CONCAT

其函数格式为：CONCAT(column1 | expression1, Column2 | expression2)，该函数用于连接两个字符串，或者连接两个列中的数据，如实例 4-20 所示。

实例 4-20　使用单行函数 CONCAT()。

```
SQL> SELECT CONCAT ('Structured Query Language','is easy to learn!')
  2  FROM dual;

CONCAT('STRUCTUREDQUERYLANGUAGE','ISEASYTO
------------------------------------------
Structured Query Languageis easy to learn!
```

在函数 CONCAT 中，参数也可以是列名，列名和表达式可以根据需要自由选择，如实例 4-21 所示。

实例 4-21　使用列名和表达式的 CONCAT 函数。

```
SQL> SELECT concat(ename,hiredate) "emp_name,hiredate"
  2  FROM emp;

emp_name,hiredate
--------------------
SMITH17-12 月-80
ALLEN20-2 月 -81
WARD22-2 月 -81
JONES02-4 月 -81
MARTIN28-9 月 -81
BLAKE01-5 月 -81
CLARK09-6 月 -81
SCOTT19-4 月 -87
KING17-11 月-81
TURNER08-9 月 -81
ADAMS23-5 月 -87

emp_name,hiredate
--------------------
JAMES03-12 月-81
FORD03-12 月-81
MILLER23-1 月 -82

已选择 14 行。
```

5. SUBSTR

其函数格式为：SUBSTR(column | expression,m [,n])，该函数从一个字符串中获取一个子串，该子串从 expression 的第 m 个字符开始，到第 n 个字符结束，如果不指定 n，则从第 m 个字符开始到 expression 表达式的结尾，如实例 4-22 所示。

实例 4-22　使用 SUBSTR 函数。

```
SQL> SELECT SUBSTR('structured query language',12)
  2  FROM dual;

SUBSTR('STRUCT
--------------
query language
```



函数 SUBSTR 在计算子串的起始位置时，一个空格占用一个字符。上述字符串'structured query language'共有 25 个字符，第 12 个字符是'q'，而起始参数 m＝12，没有指定结束字符的位置，所以默认是从第 12 个字符开始到字符串的结尾。

6. LENGTH

其函数格式为：LENGTH(column | expression)，计算字符串中的字符个数。实例 4-23 用于测试字符串'structured query language'中的字符数。

实例 4-23　使用 LENGTH 函数。

```
SQL> SELECT LENGTH('structured query language')
  2  FROM dual;

LENGTH('STRUCTUREDQUERYLANGUAGE')
---------------------------------
                               25
```

7. INSTR

其函数格式为：INSTR(column | expression, ‘string’, [,m], [n])，该函数的功能是在字符串 expression 中搜索字符串'string'，参数 m 和 n 指定搜索的开始位置和结束位置。如果没有指定 m 和 n 的值，则从字符串 expression 中搜索，如实例 4-24 所示。

实例 4-24　使用 INSTR 函数。

```
SQL> SELECT instr('structured query language','query')
  2  FROM dual;

INSTR('STRUCTUREDQUERYLANGUAGE','QUERY')
----------------------------------------
                                      12
```

该函数返回一个数字，表明字符串'query'在字符串'structured query language'中的起始位置。

8. LPAD|RPAD

其函数格式为：LPAD|RPAD (column | expression，n, ‘string’)，如实例 4-25 所示。

实例 4-25　使用 LPAD 函数。

```
SQL> SELECT LPAD(sal,10,'*')
  2  FROM emp
  3  WHERE sal >1500;

LPAD(SAL,10,'*')
--------------------
******1600
******2975
******2850
******2450
******5000
******3000

已选择 6 行。
```

在函数 LPAD(sal,10,'*')中，sal 是表 emp 中的列名，10 表示该函数的输出结果需要 10 个字符，而“*”号表示如果列 sal 的值不足 10 个字符，则在 sal 值的左边用“*”号补充。如果此时函数为 RPAD(sal,10,'*')，则在 sal 值的右边用“*”号补充。读者可以自行测试，这里不再赘述。函数 LPAD 和 RPAD 的作用就是在输出结果中增加一些补充信息，使得输出结果更具可读性。

9．TRIM

其函数格式为：TRIM (leading|trailing|both, trim_character FROM Trim_source)，该函数的作用是在字符串中剪切一个字符，输出结果是一个字符串。其参数中 leading|trailing| both 的作用分别是：函数从源字符串的头部删除要剪切的字符，还是从尾部和两边删除要剪切的字符，默认是 both，如实例 4-26 所示。

实例 4-26　使用 TRIM 函数。

```
SQL>SELECT trim ('S' FROM 'SQL is an easy Database languageS')
  2  FROM dual;

TRIM('S'FROM'SQLISANEASYDATABAS
-------------------------------
QL is an easy Database language
```

输出结果显示已经把源字符串'S' FROM 'SQL is an easy Database languageS'中的第一个 S 和最后一个 S 都删除掉。

10．REPLACE

其函数格式为：REPLACE (text,search_string,replacement_string)，该函数用于把源字符串（text）中的某个字符串（search_string）替换为另一个字符串（replacement_ string）。该函数很简单，下面给出实例 4-27，以供读者体会。

实例 4-27　使用 REPLACE 函数。

```
SQL> SELECT replace ('sql is an easy Database language','sql','Structured Query Language')
  2  FROM dual;

REPLACE('SQLISANEASYDATABASELANG
--------------------------------
Structured Query Language is an easy Database language
```

该函数将源字符串中的 sql 替换为 Structured Query Language，即用完整的英语单词更具说明性，容易理解。

4.3.2　数字型单行函数

数字型单行函数用于实现对数字的处理，其输出也是数字类型，包括如下三个函数：

- ROUND(column/expression , n)
- TRUNC(column/expression , n)
- MOD(m , n)

下面依次介绍这些函数的具体使用。

1．ROUND 函数

该函数的作用是对一个数字输出用户指定的小数位，如数字 32.1415，用户可以要求只输出小数点后的 3 位，使用该函数处理数字时应用四舍五入的规则，如实例 4-28 所示。

实例 4-28　使用 ROUND 函数。

```
SQL> SELECT round(32.1415,3)
  2  FROM dual;

ROUND(32.1415,3)
-------------------
              3.142
```

如果该函数的参数 n 为负数，则表示要求保留相应的整数位，如实例 4-29 所示。

实例 4-29　保留整数位。

```
SQL> SELECT round(32.1415,-1)
  2  FROM dual;

ROUND(32.1415,-1)
-----------------
               30
```

2．TRUNC 函数

该函数的作用是截断一个数字，只保留小数点后一定的位数，利用该函数处理数字时不使用四舍五入规则，显然 Oracle 使用截断一词的用意也是如此，如实例 4-30、实例 4-31 所示。

实例 4-30　使用 TRUNC 函数。

```
SQL> SELECT trunc (32.1414,3)
  2  FROM dual;

TRUNC(32.1415,3)
----------------
          32.141
```

实例 4-31　保留整数位。

```
SQL> SELECT trunc (32.1415,-1)
  2  FROM dual;

TRUNC(32.1415,-1)
-----------------
               30
```

3．MOD 函数

该函数的作用是求余数，如实例 4-32 所示。

实例 4-32　使用 MOD 函数（够除）。

```
SQL> SELECT mod(1000,400)
  2  FROM dual;

MOD(1000,400)
```

```
-------------
        200
```

该实例中用 1000 除以 400，商为 2，此时余数是 200，即 2*400＋200（余数）＝1000，所以经过计算之后的结果是 200。但是如果不够除应该怎么办呢，例如使用 100 除以 400 显然不够除，如实例 4-33 所示。

实例 4-33　使用 MOD 函数（不够除）。

```
SQL> SELECT mod (100,400)
  2  FROM dual;

MOD(100,400)
------------
        100
```

显然 100/400 不够除，商为 0，此时余数是 100，即 0*400＋100（余数）＝100。

4.3.3　日期型单行函数

Oracle 使用内部数字格式存储日期，默认的日期显示和输入格式为 DD-MON-RR。有效的日期从公元前 4712 年 1 月 1 日到公元 9999 年 12 月 31 日。日期在数据库中的内部存储格式为：世纪、年、月、日、时、分、秒。不论外部的日期形式如何改变，数据库对日期的内部存储格式都不会改变。

Oracle 提供了用于操作或显示日期的函数，它们包括：SYSDATE、MONTHS_BETWEEN、ADD_MONTHS、NEXT_DAY、LAST_DAY。下面依次介绍这些函数。

1．SYSDATE 函数

该函数返回系统的当前日期，该日期受操作系统限制，即 Oracle 数据库读取操作系统的时间，如实例 4-34 所示。

实例 4-34　查询 SYSDATE 的值。

```
SQL> SELECT sysdate
  2  FROM dual;

SYSDATE
---------
06-JUN-09
```

SYSDATE 函数也可以进行算数运算，例如日期函数和一个数字相加减，可以得到一个日期值，这个数字代表一个天数，如实例 4-35 所示。

实例 4-35　包含 SYSDATE 运算的查询。

```
SQL> SELECT SYSDATE + 7 , SYSDATE - 7
  2  FROM dual;

SYSDATE+7 SYSDATE-7
--------- ---------
13-JUN-09 30-MAY-09
```

两个日期型数据相减时，会得到一个数字型数据，如实例 4-36 所示。

实例 4-36 日期相减的运算查询。

```
SQL> SELECT to_date('06-JUN-10') - SYSDATE
  2  FROM dual;

TO_DATE('06-JUN-10')-SYSDATE
----------------------------
                 364.460139
```



函数 to_date()是把字符型数据转换为日期型数据的方法。上述实例得到的数字型数据表示天数，即某个日期距离当前日期还有多少天，在上例中当前日期是 06-JUN-09，而某个日期是 06-JUN-10，二者相差几乎一年。上例也验证了这个结果。

还可以在日期型数据上添加一些小时数，例如在当前日期上增加 20 个小时，得到的仍然是日期型数据。但是，此时的小时数必须除以 24，如实例 4-37 所示。

实例 4-37 在日期型数据上加小时数的查询。

```
SQL> SELECT sysdate + 20/24
  2  FROM dual;

SYSDATE+2
---------
07-JUN-09
```

此时，输出结果比当前日期多了一天，因为当前笔者的日期为 2009-6-6 13:05:51，所以加 20/24 小时后为 2009-6-7。但是如果将 20/24 改为 1/24，即只在当前日期上增加一小时，小时会改变，但是日期不会改变，如实例 4-38 所示。

实例 4-38 在当前日期上增加一小时的查询。

```
SQL> SELECT sysdate + 1/24
  2  FROM dual;

SYSDATE+1
---------
06-JUN-09
```



为了使上述日期数据的实例运行正确，需要读者设置数据库的字符集为美国英语，可使用如下指令实现。

```
SQL>alter session set NLS_DATE_LANGUAGE = 'AMERICAN';
```

2. MONTHS_BETWEEN 函数

该函数的参数为两个日期，用于得到两个日期之间的月数，即两个日期间相差几个月，如实例 4-39 所示。

实例 4-39　使用 MONTHS_BETWEEN 函数。

```
SQL>SELECT months_between('06-JUN-10','06-JUN-09'),
  2  FROM dual;
MONTHS_BETWEEN('06-JUN-10','06-JUN-09')
 -----------------------------------------------------------
                               12
```

如果函数 MONTHS_BETWEEN 中的第 1 个参数早于第 2 个参数，则得到一个负值，如实例 4-40 所示。

实例 4-40　使用 MONTHS_BETWEEN 函数。

```
SQL> SELECT months_between('06-JUN-08','06-JUN-09')
  2  FROM dual;

MONTHS_BETWEEN('06-JUN-08','06-JUN-09')
---------------------------------------
                              -12
```

3. ADD_MONTHS 函数

该函数的参数为日期型数据以及一个数字型数据 n，该函数的功能是把 n 个月添加到日期型数据上。输出结果仍为日期型数据，如实例 4-41 所示。

实例 4-41　使用 ADD_MONTHS 函数。

```
SQL> SELECT add_months(SYSDATE , 4)
  2  FROM dual;

ADD_MONTH
---------
06-OCT-09
```

系统的 SYSDATE 为 06-JUN-09，在这个日期上增加 4 个月就是 06-OCT-09。

4. NEXT_DAY 函数

该函数的参数为一个日期型数据，输出为该日期的下一个指定的日期，如实例 4-42 所示。

实例 4-42　使用 NEXT_DAY 函数。

```
SQL> SELECT next_day(sysdate,'Saturday')
  2  FROM dual;

NEXT_DAY(
---------
13-JUN-09
```

该实例是希望得到从当前日期开始的第一个 Saturday 的日期。当前日期是 06-JUN-09 星期六，所以下一个星期六为 13-JUN-09。

5. LAST_DAY 函数

该函数用于返回参数中日期的最后一天的日期，如实例 4-43 所示。

实例 4-33　使用 LAST_DAY 函数。

```
SQL> SELECT last_day(sysdate)
  2  FROM dual;

LAST_DAY(
---------
30-JUN-09
```

上述实例用于输出本月的最后一天是几号。

4.4　空值和空值处理函数

空值是非常特殊的值，既不能说它不存在，也不能说它是零。空值表示一类没有定义的值，具有不确定性。当然对于空值的运算也具有特殊性，因为具有不确定性的值是无法和具有确定性的值进行逻辑或算数运算的，Oracle 提供了多个空值处理函数，通过这些函数可实现空值（NULL）的运算。下面依次介绍什么是空值以及与空值相关的函数，这些函数包括 NVL 函数、NVL2 函数、NULLIF 函数和 COALESCE 函数等。

4.4.1　什么是空值

空值是一类没有定义的、具有不确定性的值。在数据表中，这类值无法表示，更无法显示。在 scott 用户的 EMP 表中存在空值，如实例 4-44 所示。

实例 4-44　查询 EMP 表中的空值。

```
SQL> col empno for 9999
SQL> col sal for 9999
SQL> col comm for 9999
SQL> col mgr for 9999
SQL> SELECT empno,ename,job,mgr,hiredate,sal,comm
  2   FROM emp
  3*  order by job

EMPNO ENAME      JOB         MGR HIREDATE     SAL  COMM
----- ---------- --------- ----- ---------- ----- -----
 7902 FORD       ANALYST    7566  03-12月-81  3000
 7369 SMITH      CLERK      7902  17-12月-80   800
 7900 JAMES      CLERK      7698  03-12月-81   950
 7934 MILLER     CLERK      7782  23-1月 -82  1300
 7566 JONES      MANAGER    7839  02-4月 -81  2975
 7782 CLARK      MANAGER    7839  09-6月 -81  2450
 7698 BLAKE      MANAGER    7839  01-5月 -81  2850
 7839 KING       PRESIDENT        17-11月-81  5000
 7499 ALLEN      SALESMAN   7698  20-2月 -81  1600   300
 7654 MARTIN     SALESMAN   7698  28-9月 -81  1250  1400
 7844 TURNER     SALESMAN   7698  08-9月 -81  1500     0

EMPNO ENAME      JOB         MGR HIREDATE     SAL  COMM
----- ---------- --------- ----- ---------- ----- -----
 7521 WARD       SALESMAN   7698  22-2月 -81  1250   500

已选择12行。
```

在上述输出中除了 SALSMAN 具有 COMM 佣金外，其他职位根本没有，所以在表中相应的 COMM 列的值为空值（NULL），但是这个值在表中是没有显示的。

空值可以用于表达式运算，但是因为空值不是具体的值，具有不确定性，所以下面实例 4-45 的查询不成功。

实例 4-45　查询表 EMP 中“comm=NULL”的用户数据。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal,comm
  2  FROM emp
  3  WHERE comm = NULL;

未选定行
```

通过上例可以看出，空值（NULL）不是某个值，我们可以用 NULL 表示它，但是不能直接用于计算。

那么如何实现上例中 WHERE 子句中的条件，即如何判断某列的值为空值（NULL）呢？Oracle 提供了 IS NULL 和 IS NOT NULL 运算符来处理这个运算，例如实例 4-46 使用的是 IS NULL。

实例 4-46　查询表 EMP 中“comm IS NULL”的用户数据。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal,comm
  2  FROM emp
  3  WHERE comm IS NULL;

EMPNO ENAME      JOB         MGR HIREDATE     SAL  COMM
------------- ---------------- ------------------- ----- ------- --------
 7369 SMITH      CLERK       7902  17-12 月-80    800
 7566 JONES      MANAGER    7839   02-4 月 -81   2975
 7698 BLAKE      MANAGER    7839  01-5 月 -81    2850
 7782 CLARK      MANAGER    7839  09-6 月 -81    2450
 7839 KING       PRESIDENT        17-11 月-81    5000
 7900 JAMES      CLERK       7698  03-12 月-81    950
 7902 FORD       ANALYST     7566  03-12 月-81    3000
 7934 MILLER     CLERK       7782   23-1 月 -82   1300

已选择 8 行。
```

实例 4-47 使用 IS NOT NULL 查询具有佣金的信息，即 COMM 列不为空的数据。

实例 4-47　查询表 EMP 中“comm IS NOT NULL”的用户数据。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal,comm
  2  FROM emp
  3  WHERE comm IS NOT NULL;

EMPNO ENAME      JOB         MGR HIREDATE     SAL  COMM
--------- ------------------ ---------- --------------- ----- ---------
 7499 ALLEN      SALESMAN   7698   20-2 月 -81    1600   300
 7521 WARD       SALESMAN   7698   22-2 月 -81    1250   500
 7654 MARTIN     SALESMAN   7698   28-9 月 -81    1250  1400
 7844 TURNER     SALESMAN   7698   08-9 月 -81    1500     0
```

4.4.2 NVL 函数

NVL 函数使得空值可以进行运算，它是空值转换函数。如果不使用空值转换函数，则空值是无法进行运算的。NVL 函数的语法格式如下。

```
NVL(expr1,expr2)
```

其计算规则是：如果 expr1 的值为空值（NULL），则返回 expr2 的值，否则返回 expr1 的值。其中表达式 expr1 和 expr2 的数据类型必须相同，它们可以是数值类型、字符类型和日期类型。在实例 4-48 中使用 NVL 函数计算 sal + comm 的值。

实例 4-48　使用 NVL 函数计算 sal + comm 的值。

```
SQL> SELECT empno,ename,job,mgr,hiredate,sal+NVL(comm,0)
  2  FROM emp;

EMPNO ENAME      JOB       MGR  HIREDATE    SAL+NVL(COMM,0)
----- ---------- --------- ----- ---------- ---------------
 7369 SMITH      CLERK      7902   17-12 月-80           800
 7499 ALLEN      SALESMAN  7698  20-2 月 -81          1900
 7521 WARD       SALESMAN  7698  22-2 月 -81          1750
 7566 JONES      MANAGER    7839  02-4 月 -81          2975
 7654 MARTIN     SALESMAN  7698  28-9 月 -81          2650
 7698 BLAKE      MANAGER   7839  01-5 月 -81          2850
 7782 CLARK      MANAGER   7839  09-6 月 -81          2450
 7839 KING       PRESIDENT        17-11 月-81          5000
 7844 TURNER     SALESMAN  7698  08-9 月 -81          1500
 7900 JAMES      CLERK       7698  03-12 月-81           950
 7902 FORD       ANALYST    7566   03-12 月-81          3000

EMPNO ENAME      JOB       MGR HIREDATE   SAL+NVL(COMM,0)
----- ---------- --------- ----- ---------- ---------------
 7934 MILLER     CLERK      7782    23-1 月 -82          1300

已选择 12 行。
```

从上例输出结果可以看出，COMM 列为空值的值转换为 0。如果不使用 NVL 函数进行空值转换，则无法实现计算，所以不会输出任何结果，如实例 4-49 所示。

实例 4-49　不使用 NVL 函数计算 sal + comm 的值。

```
SQL> SELECT ename,sal,comm,sal+comm
  2  FROM emp
  3  order by sal+comm;

ENAME        SAL  COMM   SAL+COMM
---------- ----- ----- ----------
TURNER      1500     0       1500
WARD        1250   500       1750
ALLEN       1600   300       1900
MARTIN      1250  1400       2650
SMITH        800
JAMES        950
MILLER      1300
FORD        3000
```

```
JONES       2975
BLAKE       2850
CLARK       2450

ENAME        SAL  COMM   SAL+COMM
---------- ----- ----- ----------
KING        5000

已选择 12 行。
```

显然，在上例输出中，由于 sal+comm 计算时 comm 列的值存在空值，所以无法计算，自然也无法显示这样的计算结果。

4.4.3　NVL2 函数

NVL2 函数是对 NVL 函数功能的增强。NVL2 函数的格式为：

```
NVL2(expr1,expr2,expr3)
```

其基本功能就是实现空值（NULL）的转换。其计算规则是：如果 expr1 为空，则返回表达式 expr3 的值，如果 expr1 不为空，则返回表达式 expr2 的值。其中 expr1 为任何数据类型，而表达式 expr2 和 expr3 为除 LONG 数据类型外的任何数据类型。

下面使用 NVL2 实现包含 sal + comm 的值，如实例 4-50 所示。

实例 4-50　使用 NVL2 实现包含 sal + comm 的值。

```
SQL> SELECT ename,nvl2(comm,sal+comm,sal)
  2  FROM emp;

ENAME      NVL2(COMM,SAL+COMM,SAL)
---------- -----------------------
SMITH                          800
ALLEN                         1900
WARD                          1750
JONES                         2975
MARTIN                        2650
BLAKE                         2850
CLARK                         2450
KING                          5000
TURNER                        1500
JAMES                          950
FORD                          3000

ENAME      NVL2(COMM,SAL+COMM,SAL)
---------- -----------------------
MILLER                        1300

已选择 12 行。
```

上例中 NVL2(comm,sal+comm,sal)的计算规则是：如果 comm 的值为空，则返回 sal 的值，如果 comm 的值不为空，则返回 sal+comm 的值。

4.4.4 NULLIF 函数

NULLIF 函数用于比较两个表达式，如果二者相等，则返回空值 NULL，如果二者不等，则返回第一个表达式的值。要求第一个表达式的值不能为 NULL。其语法格式为：

```
NULLIF(expr1,expr2)
```

实例 4-51 用于测试该函数的用法。

实例 4-51 使用 NULLIF 函数。

```
SQL> select ename,length(ename) "expr1",job,length(job) "expr2",
  2  NULLIF(length(ename),length(job)) "Comparision Result"
  3  FROM EMP;

ENAME          expr1  JOB              expr2      Comparision Result
----- ---------- --------------------------- ---------------- ----
SMITH              5  CLERK              5
ALLEN              5  SALESMAN          8               5
WARD              4  SALESMAN         8               4
JONES              5  MANAGER            7              5
MARTIN             6  SALESMAN           8              6
BLAKE              5  MANAGER          7              5
CLARK              5  MANAGER          7              5
KING                4  PRESIDENT          9               4
TURNER             6  SALESMAN          8              6
JAMES              5  CLERK              5
FORD               4  ANALYST           7               4

ENAME          expr1 JOB            expr2        Comparision Result
----- ---------- -------- ---------------- -----------------------
MILLER             6  CLERK           5               6

已选择 12 行。
```

在上例中，函数 NULLIF 包括两个表达式：一个是 length(ename)；另一个是 length(job)。函数首先比较这两个表达式的值，若二者相等则返回空值 NULL，如上例中的第 1 行记录，如果二者不等，则返回第 1 个表达式的值。

4.4.5 COALESCE 函数

COALESCE 函数用于返回该函数中第一个不为 NULL 的表达式的值。其语法格式为：

```
COALESCE(expr1,expr2,……exprn)
```

下面用实例 4-52 说明如何使用该函数。

实例 4-52 使用 COALESCE 函数。

```
SQL> SELECT ename "Employ_name",job,COALESCE(comm,1) "Comm"
  2  FROM emp
  3* ORDER BY job

Employ_nam JOB             Comm
--------------- -------------------------- ----------
```

```
FORD       ANALYST          1
SMITH      CLERK             1
JAMES      CLERK             1
MILLER     CLERK             1
JONES      MANAGER          1
CLARK      MANAGER         1
BLAKE      MANAGER         1
KING       PRESIDENT          1
ALLEN      SALESMAN        300
MARTIN     SALESMAN       1400
TURNER     SALESMAN          0

Employ_nam JOB             Comm
---------------------------------------- ---------------
WARD       SALESMAN        500

已选择 12 行。
```

在上例中，我们使用函数 COALESCE 查询雇员（employee）的佣金，如果没有佣金，则显示 1，如果佣金值不为空（NULL），则返回佣金值。上例中 JOB 为 SALESMAN 的雇员都有佣金，而其他雇员没有佣金，因为这些雇员的 COMM 列的值为 NULL。

4.5　逻辑判断功能

在高级程序设计语言中，为实现语句的逻辑结构而设计了逻辑判断语句，如 IF-THEN-ELSE。在 Oracle 的 SQL 语句中也提供了两个函数来实现逻辑判断的功能，这两个函数分别是 CASE 表达式和 DECODE 函数。

4.5.1　CASE 表达式

CASE 表达式用于逻辑判断，为了说明其用法，这里先给出其语法结构，代码如下所示：

```
CASE expr WHEN comparison_expr1 THEN return_expr1
    [WHEN comparison_expr2] THEN return_expr2
    WHEN comparison_exprn] THEN return_exprn
    ELSE else_expr]
END
```

该表达式首先比较 expr 和 comparison_expr1，如果二者相等，则返回 return_expr1，否则比较 expr 和 comparison_expr2，如果二者相等则返回 return_expr2，否则继续判断，如果都不满足，最后将返回 ELSE 后的 else_expr。实例 4-53 用于说明如何使用该表达式。

实例 4-53　使用 CASE 表达式。

```
SQL>  SELECT ename,job,sal,
  2   CASE job WHEN 'SALESMAN' THEN 1.20*sal
  3            WHEN 'MANAGER' THEN 1.30*sal
  4            WHEN 'ANALYST' THEN 1.40*sal
  5   ELSE sal END "Last Salary"
  6   FROM emp
  7  ORDER BY job;
```

```
ENAME      JOB             SAL Last Salary
-------------------- ------------------------ ---------- -----------
FORD       ANALYST        3000       4200
SMITH      CLERK           800        800
JAMES      CLERK           950        950
MILLER     CLERK          1300        1300
JONES      MANAGER        2975       3867.5
CLARK      MANAGER        2450       3185
BLAKE      MANAGER        2850       3705
KING       PRESIDENT      5000         5000
ALLEN      SALESMAN       1600       1920
MARTIN     SALESMAN       1250       1500
TURNER     SALESMAN       1500       1800

ENAME      JOB             SAL Last Salary
-------------------- ----------------------- ------------- -----------
WARD       SALESMAN       1250       1500

已选择 12 行。
```

在该实例中，对岗位为 SALESMAN、MANAGER 和 ANALYST 的雇员进行适当加薪，通过实例可以看到，这些员工的工资得到增加，通过前后对比可以很明显地看出这个变化。



在表达式 CASE 中，表达式 expr、comparison_exprn 和 return_exprn 必须是相同的数据类型，这些数据类型可能是 CHAR、VARCHAR2、NCHAR 或者 NVARCHAR2。

4.5.2 DECODE 函数

DECODE 函数与 CASE 表达式具有相同的功能，不过 DECODE 函数使用更简单，其语法格式为：

```
DECODE(col|expression, search1,result1 [,search2, result2,…,] [,default])
```

该函数的执行过程是：首先判断 search1 的值是否和 col 或 expression 的值相等，如果相等，则返回 result1，否则判断 search2 的值是否和 col 或 expression 的值相等，如果相等则返回 result2 的值，依次判断，如果都不相等，则返回默认值 default。实例 4-54 用于说明如何使用 DECODE 函数。

实例 4-54　使用 DECODE 函数。

```
SQL> SELECT ename,job,sal,
  2  DECODE(job,'SALESMAN',1.20*sal,
  3           'MANAGER',1.30*sal,
  4           'ANALYST',1.40*sal,
  5        sal)
  6  Last_Salary
  7  FROM emp
  8  ORDER BY job;

ENAME      JOB             SAL LAST_SALARY
---------- --------- ---------- -----------
```

```
FORD       ANALYST        3000       4200
SMITH      CLERK           800        800
JAMES      CLERK           950        950
MILLER     CLERK          1300       1300
JONES      MANAGER        2975     3867.5
CLARK      MANAGER        2450       3185
BLAKE      MANAGER        2850       3705
KING       PRESIDENT      5000       5000
ALLEN      SALESMAN       1600       1920
MARTIN     SALESMAN       1250       1500
TURNER     SALESMAN       1500       1800

ENAME      JOB             SAL  LAST_SALARY
---------- --------- ---------- -----------
WARD       SALESMAN       1250       1500

已选择 12 行。
```

上例的输出结果和 CASE 表达式中示例的输出结果完全一样。函数的判断过程同 CASE 表达式的判断过程也一样。

4.6　分组函数

分组函数对表中的多行数据进行操作，而每组返回一个计算结果。常用的分组函数包括：AVG、SUM、MAX、MIN、COUNT，以及 GROUP BY 子句、HAVING 子句。

这些函数的统一用法如下所示。

```
SELECT  [column,] group_function_name(column),…
FROM    tablename
[WHERE  condition]
[GROUP BY column]
[ORDER BY column];
```

4.6.1　AVG 和 SUM 函数

实例 4-55 使用 AVG 函数和 SUM 函数查询表 EMP 中员工的平均工资和所有员工的工资总和。

实例 4-55　使用 AVG 函数和 SUM 函数。

```
SQL> SELECT AVG(sal) "平均工资",SUM(sal) "总工资"
  2  FROM emp;

  平均工资     总工资
---------- ----------
2077.08333      24925
```

4.6.2　MAX 和 MIN 函数

与 AVG 和 SUM 函数不同，MAX 和 MIN 函数既可以操作数字型数据，也可以操作字符型和日期型数据，如实例 4-56、实例 4-57 所示。

实例 4-56　使用 MAX 和 MIN 函数。

```
SQL> SELECT MAX(SAL) "Highest salary", MIN(SAL) "Lowest salary"
  2  FROM emp;

Highest salary Lowest salary
-------------- -------------
          5000           800
```

实例 4-57　计算最早雇佣员工的日期和最晚雇佣员工的日期。

```
SQL> SELECT MAX(HIREDATE) "Last day", MIN(HIREDATE) "First day"
  2  FROM EMP;

Last day   First day
---------- ----------
23-1月 -82 17-12月-80
```

4.6.3　COUNT 函数

该函数用于返回经过计算得到的行数，包括空行和重复的行，如实例 4-58 用于查询表 EMP 中所有的记录个数，即表中的行数。

实例 4-58　使用 COUNT()函数。

```
SQL> SELECT count(*) "表中的行数"
  2  FROM emp;

表中的行数
----------
        12
```

实例 4-59 使用关键字 DISTINCT 返回不同的 JOB 类型数量，即去掉重复的 JOB 行记录。

实例 4-59　使用包含 DISTINCT 的 COUNT 函数。

```
SQL> SELECT count(distinct job)
  2  FROM emp;

COUNT(DISTINCTJOB)
------------------
                 5
```

从实例 4-58 和实例 4-59 可以看出，表中共有 12 行记录，包括 5 种工作职位。

4.6.4　GROUP BY 子句

在前面的内容中，使用 AVG 和 SUM 函数查询了表 EMP 中的员工平均工资和总工资数，但是如果要查询每个工作职位的员工平均工资和总工资之和又该如何计算呢？此时需要使用 GROUP BY 子句，按照工作职位分组，然后再计算，如实例 4-60 所示。

实例 4-60　使用 GROUP BY 函数。

```
SQL>SELECT JOB, AVG(sal) "平均工资",SUM(sal) "总工资"
  2  FROM emp
```

```
  3* GROUP BY JOB

JOB          平均工资     总工资
--------- ---------- ----------
ANALYST         3000       3000
CLERK     1016.66667       3050
MANAGER   2758.33333       8275
PRESIDENT       5000       5000
SALESMAN        1400       5600
```

上述查询结果是按照职位名称的顺序排序的，如果想按照总工资数的大小顺序排列，则需要使用 ORDER BY 子句，如实例 4-61 所示。

实例 4-61　使用 ORDER BY 子句。

```
SQL> SELECT JOB, AVG(sal) "平均工资",SUM(sal) "总工资
  2  FROM emp
  3  GROUP BY JOB
  4  ORDER BY "总工资";

JOB          平均工资     总工资
--------- ---------- ----------
ANALYST         3000       3000
CLERK     1016.66667       3050
PRESIDENT       5000       5000
SALESMAN        1400       5600
MANAGER   2758.33333       8275
```

显然，这样的结果可以让总工资的排序一目了然，在函数 AVG 和 SUM 后只使用别名，也能使得输出结果更加容易阅读。

分组函数可以嵌套使用，如实例 4-62 所示，即按照工作职位的分类计算最高平均工资和最低平均工资。

实例 4-62　使用分组嵌套函数。

```
SQL> SELECT MAX(AVG(sal)) ,MIN(AVG(sal))
  2  FROM EMP
  3  GROUP BY JOB;

MAX(AVG(SAL)) MIN(AVG(SAL))
------------- -------------
         5000    1016.66667
```

在执行实例 4-62 中的语句时，Oracle 首先会实现全表扫描，然后按照 JOB 分类，再计算每类的平均值，最后计算这些平均值的最大值和最小值。

4.6.5　HAVING 子句

在分组函数中，不能使用 WHERE 子句限制分组函数，所以 Oracle 设计了 HAVING 子句来执行对分组函数的某些限制。如实例 4-63 使用 HAVING 子句限制了 AVG(sal)>2000，即只显示平均工资大于 2000 的职位信息。

实例 4-63　使用 HAVING 子句。

```
SQL> SELECT job,AVG(sal)
  2  FROM emp
  3  HAVING AVG(sal)>2000
  4  GROUP BY job;

JOB        AVG(SAL)
--------- ----------
ANALYST        3000
MANAGER   2758.33333
PRESIDENT      5000
```

也可以使用 ORDRE BY 子句对 AVG(sal)进行排序，使得输出更容易阅读，如实例 4-64 所示。

实例 4-64　在分组函数中使用 ORDER BY 子句。

```
SQL>  SELECT job,AVG(sal)
  2   FROM emp
  3   HAVING AVG(sal)>2000
  4   GROUP BY job
  5  ORDER BY 2;

JOB        AVG(SAL)
--------- ----------
MANAGER   2758.33333
ANALYST        3000
PRESIDENT      5000
```



ORDER BY 2 和 ORDER BY AVG(SAL) 的效果相同，只是为了书写方便，使用数字 2 表示按照第 2 列排序。

4.7　数据操纵语言

数据操纵语言（Data Manipulation Language）用于实现对表中数据的各种操作，如向表中插入数据、删除一行数据或者更新表中的行数据。无论读者使用何种高级语言开发连接数据库的程序，数据操作语句都是使用频率最高的。下面依次介绍 INSERT 语句、UPDATE 语句和 DELETE 语句。

4.7.1　INSERT 语句

使用 INSERT 语句向表中添加一行数据的语法格式如下。

```
INSERT INTO tablename [(column [,column …] ) ]
VALUES      (value [, value… ] )
```

在上述语法格式中“()”中的“[]”号表示可选部分，即可以向表中的一列或多列插入数据。VALUES 后是插入数据的值，这些值和 tablename 后的列名一一对应。

- tablename：是要插入数据的表名字，要求用户对该表具有操作权限。
- column：是该表中的列名，用户需要向这些列插入数据，可以是一列，也可以是多列，

如果向表中的所有列插入一行数据，也可以不使用任何 column，但是需要用户清楚地知道该表中的列名和列的属性。

- values：是要插入的和列相对应的值，插入的值的数据类型必须和 column 的数据类型相匹配。

下面向 scott 用户的 dept 表中添加一行数据，即增加一行记录，其中 DEPTNO 为 50，DNAME 为 MARKETING，LOC 为 NEW YORK。为了验证添加结果，我们先查询一下表中的数据，如实例 4-65 所示。

实例 4-65　查询表 dept 中的所有数据。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

查看表 dept 中列的数据类型。

```
SQL> desc dept;
 名称                    空?      类型
 ----------------------------------------------------- --------
 DEPTNO              NOT NULL  NUMBER(2)
 DNAME                         VARCHAR2(14)
 LOC                           VARCHAR2(13)
```

其中 DEPTNO 为数字型，不允许为空（NOT NULL），DNAME 和 LOC 都为变长字符型。在插入数据时，与字符型列相对应的值可用英文输入法的单引号' '括起来。

输出结果表明当前表中有 4 行数据，我们再向表中添加一行数据，如实例 4-66 所示。

实例 4-66　向表 dept 中插入一行数据。

```
SQL> INSERT INTO dept (deptno,dname,loc)
  2  VALUES          (50,'MARKETING','NEW YORK');

已创建 1 行。
```

输入提示已经成功创建一行，下面为了验证 INSERT 语句的执行结果，再查询表 dept 中的数据。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 MARKETING      NEW YORK
```

显然，输出结果显示已经成功添加了一行数据。

如果需要向 DEPT 表中添加一行数据，但只有部门号 DEPTNO 和部门名称 DNAME，地点 LOC 还没有确定，这时可以只包含 DEPTNO 和 DNAME 两列，如实例 4-67 所示。

实例 4-67　向表 dept 中插入一行数据（没有 LOC 列对应的值）。

```
SQL> INSERT INTO dept(deptno,dname)
  2  VALUES          (60,'ACCOUNTING');

已创建 1 行。
```

再用实例 4-68 验证插入结果。

实例 4-68　查询插入结果。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 MARKETING      NEW YORK
        60 ACCOUNTING

已选择 6 行。
```

在当前表中新增了一行记录，其中部门号 DEPTNO 为 60，部门名字为 ACCOUNTING，而部门所在地没有值，Oracle 使用空值（NULL）填充，此时读者也可以使用实例 4-69 得到同样的插入效果。

实例 4-69　向表 dept 中插入一行数据。

```
SQL> INSERT INTO dept (deptno,dname,loc)
  2  VALUES          (60,'ACCOUNTING',NULL);

已创建 1 行。
```

还有一种插入方式，即从另一张表复制数据，这种方式的语法结构如下。

```
INSERT INTO tablename [(column [,column …] ) ]
SELECT column [,column…]
FROM   another_tablename
WHERE clause
```

4.7.2　UPDATE 语句

UPDATE 语句用于更新表中的数据，如在表 dept 中，需要把刚插入的记录的 LOC 部门所在地设置为 NEW YORK。此时就需要 UPDATE 语句更新表中的该行记录。其语法格式如下。

```
UPDATE   tablename
SET       column = value [, column = value, … ]
[WHERE   condition];
```

语法解释如下。

- tablename：更新的表名。
- column：更新的列。
- value：更新的列的值。
- condition：通过条件限制要更新的列所在的行。

在实例 4-67 中我们在表 dept 中新增了一行记录，DEPTNO 为 60，DNAME 为 ACCOUNTING，但是没有确定 LOC 部门所在地。我们把部门所在地设置为 NEW YORK，通过实例 4-70 说明如何使用 UPDATE 语句。

实例 4-70　使用 UPDATE 语句更新表 DEPT 中的数据。

```
SQL> UPDATE dept
  2  SET  LOC ='NEW YORK'
  3  WHERE DEPTNO = 60;

已更新 1 行。
```

查询更新结果，如实例 4-71 所示。

实例 4-71　查询实例 4-70 的更新结果。

```
SQL> SELECT *
  2  FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 MARKETING      NEW YORK
        60 ACCOUNTING     NEW YORK

已选择 6 行。
```

查询结果显示更新结果正确，在实例 4-70 中 LOC 的值是直接给出的，Oralce 也允许使用一个子查询来赋予 LOC 值。这里给出其语法结构，读者可自行尝试。

```
UPDATE table
SET     column =
               (SELECT column
                FROM tablename
                WHERE conditon)
       [,
column =
                (SELECT column
                FROM tablename
                WHERE conditon)]
[WHERE condition];
```

4.7.3 DELETE 语句

DELETE 语句用于删除不需要的记录，该语句使用较简单，其语法格式如下所示。

```
DELETE [FROM]  tablename
[WHERE          condition];
```

语法解释如下。

- tablename：删除的数据所在的表名。
- condition：限制要删除的行，该条件可以是指定具体的列名、表达式、子查询或者比较运算符。

下面我们删除表 dept 中 DEPTNO 为 60 的记录，如实例 4-72 所示。

实例 4-72　删除表 dept 中 DEPTNO 为 60 的记录。

```
SQL> DELETE FROM dept
  2  WHERE DEPTNO = 60;

已删除 1 行。
```



在 DELETE 语句中的 FROM 关键字是可选的，使用 FROM 关键字更合乎英语的语法习惯，更容易记忆。WHERE 子句也是可选的。如果不使用 WHERE 子句，将删除表中的所有行。

4.8　本章小结

本章对 SQL 语句进行了概述。通过 SQL 语句中的简单查询语句，使得读者对 SQL 语句、算数运算、别名及 DISTINCT 运算有了直观的认识。在书写 SQL 语句时，需要注意书写规范。函数增强了 SQL 语句的功能，使得大量的运算得以简化，本章简单介绍了单行函数和分组函数，熟练使用这些函数对于读者使用 SQL 语句很有帮助。数据操作语句也是 SQL 语句中经常使用的，如插入数据 INSERT、更新数据 UPDATE、删除数据 DELETE 等。本章还着重介绍了空值 NULL 的概念，及其如何操作空值的计算。读者需要很好地理解空值 NULL，并熟练掌握空值的相关运算。

第 5 章

PL/SQL编程基础

本章将重点介绍 PL/SQL 的编程基础，这些基础包括十分重要的 PL/SQL 变量类型，任何编程语言都需要知道它可以操作的数据类型，理解这些数据类型以及数据类型的特点是程序员必备的基础。同时我们将介绍如何初始化定义的变量，这些知识可以确保我们能够编写出基础的 PL/SQL 程序。

5.1 数据类型

任何高级编程语言都必须知道我们可以操纵的数据类型，因为编程除了复杂的逻辑结构外，最终还是对数据的处理，知道语言环境所支持的数据类型是非常重要的。PL/SQL 的数据类型包含了 SQL 的数据类型，但是两者之间也有区别，下面先学习 PL/SQL 的常用数据类型并分析其用法。

5.1.1 CHAR 和 VARCHAR2 数据类型

VARCHAR2 是变长的字符数据类型，该变量的定义必须使用字面数值参数，字符的最大长度为 32767 字节。而 VARCHAR2 数据类型对应的列的最大宽度为 4000 字节。

变量长度的定义要符合实际的数据需求，如果定义的变量长度尺寸过小，则系统 PL/SQL 引擎会报错，如实例 5-1 所示。

实例 5-1 变量长度错误。

```
SQL> declare
  2   var_test varchar2(3);
  3  begin
  4   var_test:='TESTTEST';
  5  end;
  6  /
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 4
```

上面我们定义了一个变量 var_test，数据类型为 varchar2(3)，变量长度为 3，而我们在 PL/SQL 执行阶段赋予了该变量一个超过 3 个字符长度的字符串，显然，字符串缓冲区太小，系统报错。

那么，如果我们创建了一个表，表中 student_name 字段的数据类型为 varchar2(3)，而使用

PL/SQL 过程向该表插入数据时，使用了声明部分定义的变量 var_name，数据类型为 varchar2(20)。通过在 PL/SQL 过程中为变量 var_name 赋值，从而通过这个赋值后的变量将数据插入表中，此时同样会报错，如实例 5-2 所示，分析之后，我们再改正这段程序。

实例 5-2　重新修改程序。

```
SQL> drop table student;
create table student (student_id number,student_name varchar2(3));
declare
   var_id   number := &id;
   var_name varchar2(20) := '&name';
begin
 insert into student values(var_id,var_name);

 end;
/
Table dropped.

SQL>
Table created.

SQL> /

Enter value for id: 1
old   2:    var_id   number := &id;
new   2:    var_id   number := 1;
Enter value for name: tomtomtotmtot
old   3:    var_name varchar2(20) := '&name';
new   3:    var_name varchar2(20) := 'tomtomtotmtot';
declare
*
ERROR at line 1:
ORA-12899: value too large for column "SYS"."STUDENT"."STUDENT_NAME" (actual:
13, maximum: 3)
ORA-06512: at line 5
```

此时，我们将字面值'tomtomtotmtot'赋予了变量 var_name，占据 13 个字符长度，而我们定义的表中的 student_name 列只有 3 个字符长度，无法容纳，所以报错，下面修改一下这个程序，只须改变表的数据类型长度即可，如实例 5-3 所示。

实例 5-3　修改表的数据类型长度。

```
SQL> drop table student;
create table student (student_id number,student_name varchar2(30));
Table dropped.

SQL>

Table created.
```

删除表，再重建表，然后执行插入数据的 PL/SQL 过程，如实例 5-4 所示。

实例 5-4　删除表、重建表，再插入数据。

```
SQL> declare
   var_id   number := &id;
```

```
   var_name varchar2(20) := '&name';
begin
 insert into student values(var_id,var_name);

 end;
/
Enter value for id: 1
old   2:    var_id   number := &id;
new   2:    var_id   number := 1;
Enter value for name: tomtomtotmtot
old   3:    var_name varchar2(20)  := '&name';
new   3:    var_name varchar2(20)  := 'tomtomtotmtot';

PL/SQL procedure successfully completed.
```

此时该过程执行成功，因为表中的数据类型长度可以容纳该对象大小。使用 CHAR 和 VARCHAR2 定义变量长度时要注意字符集问题，如果是单字节字符集，则最大尺寸为 32767 个字符长度，如果是 n 字节字符集，则最大尺寸为 32767/n 个字符长度。

CHAR 数据类型存储定长的字符数据，如果赋予该变量的字面值较小，则使用空格填充，该变量定义时参数可选，默认为 1 个字节，如果使用参数，则最大为 2000 个字节。下面我们通过实例 5-5 进一步体会如何使用该变量和应注意的问题。

实例 5-5　使用 CHAR 类型变量。

```
SQL> declare
  var_char1  char;
  var_char2  char(20);
begin
  var_char1 := 'a';
  var_char2 := 'abc';

  dbms_output.put_line ('var_char1 is : '||var_char1||'*');
  dbms_output.put_line ('var_char2 is : '||var_char2||'*');
end;
.
/
```

在上述 PL/SQL 语句块中，我们定义了变量 var_char1 的数据类型为 char，采用默认参数设置，默认为 1 个字节大小，变量 var_char2 的数据类型为 char(20)，参数为 20。如果赋予该变量的值不能填充这个大小，则使用空格填充。接着在语句块的执行部分，我们为变量 var_char1 赋予一个字符'a'，为变量 var_char2 赋予 3 个字符，显然在第 2 个赋值操作后，变量 var_char2 需要使用空格填充。我们通过执行该 PL/SQL 过程验证分析结果，执行结果如下所示。

```
var_char1 is : a*
var_char2 is : abc                 *

PL/SQL procedure successfully completed.
```

我们在输出时使用'*'以更直观的方式显示空格效果，显然变量 var_char1 不需要填充，而变量 var_char2 需要使用 17 个空格填充。

CHAR 的这个特性也提醒我们，在编程时，如果已经明确知道字符长度，则可以使用 CHAR 类型，如果字符长度变化不定则最好使用 VARCHAR2 数据类型，因为该类型的数据长度是不可变的。

5.1.2 NUMBER 数据类型

NUMBER 数据类型可以存储任何大小的定点和浮点数，定义该变量的语法为 number(precision,scale)，即需要定义数据精度和数值范围，这些参数必须使用字面值实现。精度的最大值为 38 个十进制位，数值范围从 0~127，根据数值范围进行数据的四舍五入，数值范围可以为负数。下面我们演示该数据类型如何使用，如实例 5-6 所示。

实例 5-6 使用 NUMBER 数据类型。

```
SQL> declare
  var_num1  number(5,2);
  var_num2  number(4,-3);
  var_num3  number;
begin
  var_num1 := 123.567;
  var_num2 := 4563.5;
  var_num3 := 1234.567;
 dbms_output.put_line('var_num1 : '||var_num1);
 dbms_output.put_line('var_num2 : '||var_num2);
 dbms_output.put_line('var_num3 : '||var_num3);
end;
.
/  SQL>
```

上面我们定义了 3 个变量，对应的都是 NUMBER 数据类型，但是使用的精度以及数值范围不同，变量 var_num1 定义为 number(5,2)，即数字的位数为 5，2 表示数值范围，即数据会在最近的百分位完成四舍五入的操作。变量 var_num2 定义为 number(4,-3)，数字 4 表示数据的位数为 4，-3 表示会在千分位进行四舍五入，变量 var_num3 定义为 number，采用默认的设置。

下面是该 PL/SQL 过程的输出结果。

```
var_num1 : 123.57
var_num2 : 5000
var_num3 : 1234.567

PL/SQL procedure successfully completed.
```

我们赋予的三个变量的值如下所示。

```
  var_num1 := 123.567;
  var_num2 := 4563.5;
  var_num3 := 1234.567;
```

对于 var_num1 := 123.567，因为数据类型的参数限制，只能保留 5 位数字，在百分位完成四舍五入，所以最后取值为 123.57。对于 var_num2 := 4563.5，因为数据类型的参数限制，只能保留 4 位数字，在千分位完成四舍五入，所以最后取值为 5000。对于 var_num3 := 1234.567，我们采用默认设置，所以直接输出该值。

5.1.3 LONG 和 LONG RAW 数据类型

LONG 数据类型用于存储变长的字符串，LONG 值的最大长度为 2GB，这是与 VARCHAR2

的唯一区别，显然由于 LONG 类型具有容量优势，因此可以使用 LONG 类型变量存储文本、字符数组以及各种文档，可以使用 DML 语句操作 LONG 列。下面我们创建一个表并插入一些字符数据，通过查询插入的数据来演示 LONG 的存储效果，如实例 5-7 所示。

实例 5-7　创建一个表并插入一些字符数据。

```
SQL> create table tlong (id number,contents long);

Table created.
SQL> insert into tlong values (1,'Oracle Database PL/SQL Language Reference,
11g Release 2 (11.2)');

1 row created.
```

以上代码创建了表 tlong，列 contents 使用 LONG 数据类型存储长文本，然后向表中插入了一行字符。下面使用 select 查询 LONG 类型的数据列，如实例 5-8 所示。

实例 5-8　查询 LONG 类型的数据列。

```
SQL> select * from tlong;

    ID  CONTENTS
------------ -------------------------------------------------------
    1    Oracle Database PL/SQL Language Reference, 11g Release 2 (11.2)
```

对于 LONG 类型的数据列，可以使用常规的 DML 语句完成数据的 SELECT、UPDATE 以及 INSERT 操作，但是使用 LONG 数据类型的列的操作还是存在限制，即不能在表达式、函数以及如 WHERE、GROUP BY 等特定语句中引用 LONG 类型的数据列，如实例 5-9 所示。

实例 5-9　应用 WHERE、GROUP BY 等特定语句。

```
SQL> select * from tlong order by contents;
select * from tlong order by contents
      *
ERROR at line 1:
ORA-00997: illegal use of LONG datatype


SQL> select * from tlong group by contents;
select * from tlong group by contents
                         *
ERROR at line 1:
ORA-00997: illegal use of LONG datatype

SQL> select * from tlong where contents='Oracle';
select * from tlong where contents='Oracle'
                        *
ERROR at line 1:
ORA-00997: illegal use of LONG datatype
```

显然，从输出中可以知道 Oracle 限制了 LONG 类型数据列的上述操作，提示为 LONG 数据类型的 illegal 使用。

LONG RAW 用于存储原始的二进制变量数据，最大值为 2GB。这样的数据需要使用 Oracle 的包来读取并转化为二进制数据，然后存储，并使用相反的方法来读取，不能使用 DML 语句来直

接查询或进行其他操作。

5.1.4 BOOLEAN 数据类型

布尔数据类型用于存储 TRUE、FALSE 以及 NULL。显然 TRUE 表示布尔真值，FALSE 表示布尔假值。而 NULL 表示这个值是未知的，不代表任何意义。BOOLEAN 类型的数据只能赋予 TRUE、FALSE 和 NULL 三个值，并且 SQL 没有数据类型与 BOOLEAN 值相对应。

下面创建一个表，如实例 5-10 所示。

实例 5-10 创建表。

```
SQL> create table t (id number,name varchar2(20),b boolean);
create table t (id number,name varchar2(20),b boolean)
                                              *
ERROR at line 1:
ORA-00902: invalid datatype
```

显然，系统提示没有该数据类型。同样不能使用 DBMS_OUTPUT.PUT_LINE 过程，如实例 5-11 所示。

实例 5-11 使用 DBMS_OUTPUT.PUT_LINE 过程。

```
SQL> set serveroutput on;
declare
 var_b  boolean :=true;
begin
 dbms_output.put_line(var_b);
end;
/SQL>
 dbms_output.put_line(var_b);
 *
ERROR at line 4:
ORA-06550: line 4, column 2:
PLS-00306: wrong number or types of arguments in call to 'PUT_LINE'
ORA-06550: line 4, column 2:
PL/SQL: Statement ignored
```

下面我们测试一下 BOOLEAN 类型。但是 BOOLEAN 的值可以通过 IF 或者 CASE 语句完成转化，如实例 5-12 所示。

实例 5-12 测试一下 BOOLEAN 类型。

```
SQL> CREATE PROCEDURE print_boolean (b BOOLEAN)
AS
 BEGIN
CASE
WHEN b IS NULL THEN DBMS_OUTPUT.PUT_LINE('Unknown');
WHEN b THEN DBMS_OUTPUT.PUT_LINE('Yes');
WHEN NOT b THEN DBMS_OUTPUT.PUT_LINE('No');
END CASE;
END;
/

Procedure created.
```

上面我们创建了一个存储过程 print_boolean，以演示使用 CASE 语句完成对 BOOLEAN 类型数据的直接使用。该存储过程的参数是一个布尔值，参数名为 b。在调用该存储过程时，可根据 b 的值调用不同的 case 语句：如果 b 为 null，则输出 Unknown；如果 b 为 true，则输出 Yes；如果 b 为 false，则输出 No。

下面将 3 次调用该存储过程，如实例 5-13 所示。

实例 5-13　3 次调用该存储过程。

```
SQL> BEGIN
print_boolean(TRUE);
print_boolean(FALSE);
print_boolean(NULL);
END;
/
Yes
No
Unknown

PL/SQL procedure successfully completed.
```

5.1.5　PLS_INTEGER 数据类型

在 Oracle 的 PL/SQL 语言环境中，PLS_INTEGER 数据类型与 BINARY_INTEGER 是相同的，这里我们统一使用 PLS_INTEGER 数据类型，该数据类型是带符号的整数数据类型，有效范围是（在 32 位机器上）：-2,147,483,648~2,147,483,648。

因为 PLS_INTEGER 直接使用硬件存储和计算，所以相对于 NUMBER 数据类型，PLS_INTEGER 数据类型具有存储空间小以及计算快速的优势。下面我们通过实例 5-14 演示 NUMBER 数据类型和 PLS_INTEGER 数据类型各自执行的计算时间。

实例 5-14　比较计算时间。

```
SQL> / SQL> set serveroutput on;
SQL> declare
  2  var_num         number :=1 ;
  3  var_plsinteger  pls_integer := 1;
  4
  5  var_start       number;
  6  var_end         number;
  7
  8  begin
  9   var_start :=DBMS_UTILITY.get_time;
 10   for i in 1..60000000 loop
 11      var_num := var_num+1;
 12   end loop;
 13   var_end :=DBMS_UTILITY.get_time;
 14   dbms_output.put_line('number type '||(var_end-var_start));
 15
 16   var_start :=DBMS_UTILITY.get_time;
 17   for i in 1..60000000 loop
 18      var_plsinteger := var_plsinteger+1;
 19   end loop;
 20   var_end :=DBMS_UTILITY.get_time;
```

```
 21   dbms_output.put_line('plsinteger type '||(var_end-var_start));
 22* end;
```

在上例中，我们定义了两个变量 var_num 和 var_plsinteger，数据类型分别为 NUMBER 和 PLS_INTEGER。对这两个变量执行相同次数的计算后，二者的执行时间具有什么区别呢？下面是执行结果。

```
number type 666
plsinteger type 220

PL/SQL procedure successfully completed.
```

显然，NUMBER 数据类型的变量计算消耗的时间为 666，而 PLS_INTEGER 数据类型的变量计算消耗的时间为 220。后者的计算速度要快得多。

但使用 PLS_INTEGER 时存在溢出问题，如实例 5-15 所示。

实例 5-15　存在溢出问题。

```
DECLARE
p1 PLS_INTEGER := 2147483647;
p2 PLS_INTEGER := 1;
n NUMBER;
BEGIN
n := p1 + p2;
END;
/
Result:
DECLARE
*
ERROR at line 1:
ORA-01426: numeric overflow
ORA-06512: at line 6
```

为了防止溢出问题，我们可以通过定义 INTEGER 数据类型的变量来解决，如实例 5-16 所示。

实例 5-16　防止溢出问题。

```
DECLARE
p1 PLS_INTEGER := 2147483647;
p2 INTEGER := 1;
n NUMBER;
BEGIN
n := p1 + p2;
END;
/
Result:
PL/SQL procedure successfully completed.
```

SIMPLE_INTEGER 是 PLS_INTEGER 的子类型，具有非空约束的优势，如果指定了 NULL 值则系统会报错，在实际编程中，如果对于整数数据类型需要非空约束，则使用 SIMPLE_INTEGER 可以减少一个判断步骤，也减少了计算量，如实例 5-17 所示。

实例 5-17　使用 SIMPLE_INTEGER。

```
SQL> l
  1  DECLARE
```

```
  2  a SIMPLE_INTEGER;
  3  b PLS_INTEGER := Null;
  4  BEGIN
  5  a := b;
  6* END;
SQL> /
a SIMPLE_INTEGER;
  *
ERROR at line 2:
ORA-06550: line 2, column 3:
PLS-00218: a variable declared NOT NULL must have an initialization assignment
```

因为数据类型 SIMPLE_INTEGER 具有非空约束，所以在定义时必须赋予初始值，在上例中，我们在声明部分没有为变量 a 赋值，所以提示错误。即使赋值，如果在程序执行过程中为该变量赋予 NULL 值，也会报错，如实例 5-18 所示。

实例 5-18　赋予 NULL 值后同样报错。

```
DECLARE
a SIMPLE_INTEGER := 1;
b PLS_INTEGER := NULL;
BEGIN
a := b;
END;
/
Result:
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 5
```

5.1.6　DATE 和 TIMESTAMP 数据类型

DATE 为日期类型，用于存储定长的日期值，该变量的有效范围在 January 1 4712 BC 和 December 31 9999 AD 之间。存储日期值的默认值是当月的第一天，时间部分（时、分、秒）的默认值是零时。存储日期时需要注意字符集的问题，如果希望显示中文格式的日期，而存储的是英语格式的日期，则需要设置相应参数，以使其显示满足要求，如实例 5-19 所示。

实例 5-19　应用 DATE 数据类型。

```
SQL>  declare
    var_date1 date;
    var_date2 varchar2(20);
   begin
     select sysdate into var_date1 from dual;
     dbms_output.put_line('var_date1 is :'||var_date1);
    select to_char(sysdate,'yyyy-mm-dd hh24:mi:ss') into var_date2 from dual;
    dbms_output.put_line('var_date2 is :'||var_date2);
    end;
/
var_date1 is :16-DEC-11
var_date2 is :2011-12-16 04:48:18

PL/SQL procedure successfully completed.
```

在上例中，我们定义了 DATE 类型的变量 var_date1，在可执行程序段内提取出当前的日期为该变量赋值。下面我们定义一个 TIMESTAMP 变量，如实例 5-20 所示。

实例 5-20 应用 TIMESTAMP 数据类型。

```
SQL> declare
  2  hire_date timestamp;
  3  begin
  4   select sysdate into hire_date from dual;
  5     dbms_output.put_line('hire_date :' ||hire_date) ;
  6* end;
SQL> /
hire_date :16-DEC-11 04.33.32.000000 AM

PL/SQL procedure successfully completed.
```

在上例中我们定义了一个 TIMESTAMP 数据类型，在执行部分为其赋值并在屏幕上打印该值，从输出可以看出使用 TIMESTAMP 的时间更加准确，显示当前的时间是 16-DEC-11 04.33.32.000000 AM。

5.1.7 ANCHORED 数据类型

在 PL/SQL 数据类型中 ANCHORED 数据类型是十分灵活的一种数据类型。ANCHORED 数据类型基于底层的数据库对象，因此一旦定义为该类型，它会随着数据对象中数据类型的变化而作相应变换，从而增加了程序的灵活性，用户不必修改程序就可以适应底层数据类型的改变。ANCHORED 数据类型的语法格式如下所示：

```
变量名  对象类型属性%TYPE。
```

如果我们定义与表 employees 的列 first_name 和 salary 相同的数据类型，则可以进行如下定义：

```
 var_name  employees.first_name%type;
 var_salary  employees.salary%type;
```

下面我们编写一个具体的匿名过程来测试该数据类型。变量定义如上所示，下面我们读取表 employees 中的 employee_id 为 206 的员工的 first_name 和 salary 信息，并存储到我们上面定义的变量中，如实例 5-21 所示。

实例 5-21 测试 ANCHORED 数据类型。

```
SQL> set serveroutput on;
SQL> declare
  2   var_name  employees.first_name%type;
  3   var_salary  employees.salary%type;
  4  begin
  5    select first_name,salary into var_name,var_salary
  6    from employees where employee_id=206;
  7    dbms_output.put_line('var_name is :'||var_name);
  8    dbms_output.put_line('var_salary is :'||var_salary);
  9* end;
SQL> /
var_name is :William
var_salary is :9000
```

```
PL/SQL procedure successfully completed.
```

设置后，变量 var_name 具有与表 employees 中的列 first_name 相同的数据类型，var_salary 具有与表 employees 中的列 salary 相同的数据类型。此时如果表 employees 的 first_name 数据类型改变，则不需要修改匿名过程的变量定义，而是可以直接使用该匿名过程完成相同的功能。这样就使得数据库对象的变化对用户程序透明。

ANCHORED 的数据类型与表中的数据类型一样，用户可以更加灵活地控制数据类型，不需要知道表中的数据类型是什么，具体的数据类型可在编译时确定。但是必须保证这个表是存在的，否则即使编译成功，在执行时也会报错。

5.1.8　自定义数据类型

除了 PL/SQL 的数据类型外，Oracle 也允许用户自行定义数据类型，这些数据类型为 PL/SQL 数据类型的子类型，用户定义的这些数据类型可以是无约束的子类型，也可以是有约束的子类型。下面我们分别介绍这些用户定义的子类型的方法和示例。

1．无约束的用户定义子类型

无约束的用户定义子类型与基类型具有相同的取值范围，所以实际上它是基类型的别名。在无约束用户定义子类型与其基类型之间可以互换使用，二者之间不会发生数据类型转换。无约束用户定义子类型的定义语法如下所示。

```
SUBTYPE  子类型名称  IS 基类型。
```

用户定义的子类型同样需要在 DECLARE 部分声明，通过下面的实例 5-22 进行演示。

实例 5-22　无约束的用户定义子类型。

```
SQL>set serveroutput on;
SQL>declare
  2    subtype mynumber  is number;
  3      var_num1   mynumber(6,2):=100;
  4      var_num2   mynumber(8,2):=200;
  5      var_num3   mynumber(8,2):=300;
  6      var_max_num constant mynumber(8,2):=300000.00;
  7   subtype mynatural is natural;
  8      var_nat1 mynatural :=1;
  9      var_nat2 mynatural :=2;
 10      var_nat3 mynatural :=3;
 11  begin
 12    var_num1 := var_num1+var_nat1;
 13    var_num2 := var_num2+var_nat2;
 14    var_num3 := var_num3+var_nat3;
 15     dbms_output.put_line('var_num1 :'||to_char(var_num1));
 16     dbms_output.put_line('var_num2 :'||to_char(var_num2));
 17     dbms_output.put_line('var_num3 :'||to_char(var_num3));
 18*  end;
```

在上例中，我们自定义了两个数据类型：一个是 mynumber，另一个是 mynatural。对于 mynumber 数据类型，它的基类型是 number，显然我们定义的数据类型与基类型相同，取值范围也相同，所

以我们就可以使用 mynumber 定义自己需要的变量。下面是执行上例的结果。

```
SQL> /
var_num1 :101
var_num2 :202
var_num3 :303

PL/SQL procedure successfully completed.
```

2. 有约束的用户定义子类型

有约束的用户定义子类型，同样需要通过基类型定义，但是这个用户定义数据类型具有约束限制，也就是说用户在定义时就已经给出了该数据类型的范围、非空等约束。有约束的用户定义子类型的定义如下所示。

```
SUBTYPE subtype_name IS base_type
{ precision [, scale ] | RANGE low_value .. high_value } [ NOT NULL ]
```

在上述语法中，我们允许使用精度、范围或者约束来定义用户定义子类型。下面通过实例 5-23 进行说明。

实例 5-23　有约束的用户定义子类型。

```
DECLARE
SUBTYPE Balance IS NUMBER(8,2);
checking_account Balance;
savings_account Balance;
BEGIN
checking_account := 2000.00;
savings_account := 1000000.00;
END;
```

在上例中，我们使用精度定义了用户定义数据类型 Balance，此时，我们在程序块的执行部分为用户定义数据类型的变量赋值，但是显然第 2 个变量 savings_account 不满足我们的约束要求，在执行时会报错，执行该过程时的错误提示如下所示。

```
Result:
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 9
```

错误解释 ORA-06502 说明上述代码违反了约束的具体内容。

在实例 5-24 中，我们定义了三个用户定义子类型：UNDER10、BETWEEN1099、Under100，三者的范围不同，具体范围如下例粗体部分所示。由于我们定义的三个数据类型的范围不同，所以在我们赋值时会发生潜在的数据转换，但是如果不满足约束条件限制，则会触发异常。

实例 5-24　不满足约束条件限制。

```
SQL>DECLARE
  2  SUBTYPE UNDER10 IS PLS_INTEGER RANGE 0..9;
  3  SUBTYPE BETWEEN1099 IS PLS_INTEGER RANGE 10..99;
  4  SUBTYPE Under100 IS PLS_INTEGER RANGE 0..99;
  5      d10     UNDER10 := 4;
```

```
  6      d1099   BETWEEN1099:= 35;
  7      u100    Under100;
  8   BEGIN
  9      u100  := d10;
 10      u100  := d1099;
 11      d1099  := d10;
 12*   END;
SQL> /
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 11
```

在上例中我们在可执行部分为三个自定义类型的变量赋值，此时会发生潜在的转换，这涉及到了三个语句，代码如下所示：

```
  9      u100  := d10;
 10      u100  := d1099;
 11      d1099  := d10;
```

在语句 9 中，由于 d10 的值为 4，而其取值范围为 0~100 的变量 u100，所以不会报错，在语句 10 中，由于 d1099 的值为 35，所以，它赋予取值范围是 0~100 的变量 u100 也不会报错。但是在语句 11 中，d10 的值为 4，而将它赋予取值范围为 10~99 的变量，显然这样的赋值无法转换，所以触发 ORA-06502 错误。

5.2　保留字

保留字是 PL/SQL 中专门使用的关键字，如 DECLARE、BEGIN、END、EXCEPTION 等。这些关键字不能在 PL/SQL 代码块中使用，例如不能把保留字作为变量或者其他进行定义，如实例 5-25 所示。

实例 5-25　错误应用保留字。

```
SQL> set serveroutput on;
declare
 begin date  := sysdate;
 end   date  :=sysdate+1;
begin
  dbms_output.put_line('beginning time  :  '||begin);
  dbms_output.put_line('end       time  :  '||end);
end;
.
/
```

在上例中，我们使用保留字 BEGIN 和 END 定义了 DATE 类型的两个变量，并且使用 DBMS_OUTPUT 包的 PUT_LINE 过程在屏幕上打印这个变量值。下面我们执行这个 PL/SQL 语句块。执行结果如下所示。

```
 end   date  :=sysdate+1;
             *
ERROR at line 3:
```

```
ORA-06550: line 3, column 13:
PLS-00103: Encountered the symbol"="when expecting one of the following:
;
ORA-06550: line 5, column 45:
PLS-00103: Encountered the symbol "BEGIN" when expecting one of the following:
( - + case mod new null <an identifier>
<a double-quoted delimited-identifier> <a bind variable>
continue avg count current max min prior sql stddev
ORA-06550: line 6, column 45:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
( - + case mod new null <an identifier>
<a double-quoted delimited-identifier> <a bind variable>
continue avg count current max min prior sql stddev su
```

输出说明我们对保留字是非法使用，显然，保留字 BEGIN 和 END 对于 PL/SQL 引擎来说是非法字符，所以不能用于定义变量，因此编译器将输出大量的错误信息。

5.3 变量

在 PL/SQL 编程中，PL/SQL 语句块中的变量必须提前声明才能被程序使用。

5.3.1 变量的定义与初始化

声明的变量可以赋予初始值、指定非空约束，也可以使用默认值，或者定义为常量，常量在整个语句块中不会发生变化并且必须在声明时赋值。

下面我们给出一个实例来说明变量未声明时引起的错误，如实例 5-26 所示。

实例 5-26 变量未声明时引起的错误。

```
SQL> set serveroutput on;
declare
   var_manager_id number :=&manager_id;
begin
  select count(*) into sum from employees where manager_id=var_manager_id;
end;
.
/
SQL>
Enter value for manager_id: 122
old   2:    var_manager_id number :=&manager_id;
new   2:    var_manager_id number :=122;
  select count(*) into sum from employees where manager_id=var_manager_id;
                       *
ERROR at line 4:
ORA-06550: line 4, column 24:
PLS-00201: identifier 'SUM' must be declared
ORA-06550: line 4, column 28:
PL/SQL: ORA-00904: : invalid identifier
ORA-06550: line 4, column 3:
PL/SQL: SQL Statement ignored
```

在上例中，我们定义了一个变量 var_manager_id，该变量使用替代变量&manager_id，在 PL/SQL

语句块执行时才绑定该变量的值。从表 employees 中计算指定 manager_id 的员工数量，即某个部门的员工数量，并存入一个变量 sum 中，但是该变量没有声明，因此系统报错。

下面我们再通过一个计算圆周周长的 PL/SQL 程序演示如何声明或者初始化一个变量，如实例 5-27 所示。

实例 5-27　声明或者初始化一个变量。

```
 set serveroutput on;
declare
 circle  number(10,2);
 radius  number(8,2) :=&r;
 pai   constant  number  := 3.14;
begin
 circle:= 2*pai*radius;
 dbms_output.put_line('circle is : '||circle);
end;
.
/
```

在上例中，我们在 DECLARE 部分声明了三个变量：一个是 circle（用来存储圆周的周长，使用默认值）；一个是 radius（定义了圆周的半径，该值使用替代变量，在执行时由用户输入）；一个是 pai（我们定义该值为常量，初始值为 3.14，该值在 PL/SQL 程序的整个生命周期中保持不变）。下面执行这个程序，这里要注意常量的定义如下：

```
变量名  constant 保留字 数据类型 赋值语句。
```

接下来我们执行该 PL/SQL 程序。

```
Enter value for r: 2
old   3:  radius  number(8,2) :=&r;
new   3:  radius  number(8,2) :=2;
circle is : 12.56

PL/SQL procedure successfully completed.
```

输入圆周的半径为 2，计算结果：圆周周长为 12.56。

变量的声明可以采用默认值，那么这些默认值是什么呢？下面我们测试一下该问题，以了解不同数据类型的变量默认值，如实例 5-28 所示。

实例 5-28　变量默认值。

```
SQL> set serveroutput on;
 declare
  first_name       varchar2(20);
  hire_date   date;
  id         number;
  last_name   employees.last_name%type;
begin
 dbms_output.put_line('first_name :'||first_name);
 dbms_output.put_line('hire date  :'||hire_date);
 dbms_output.put_line('id         :'||id);
 dbms_output.put_line('last_name  :'||last_name);
end;
.
/
```

在上例中，我们定义了 4 个变量，first_name 变量的数据类型为 varchar2，hire_date 变量的数据类型为 date，id 变量的数据类型为 number，last_name 变量的数据类型为 anchored。下面是该语句的执行结果。

```
SQL> /
first_name :
hire date  :
id         :
last_name  :

PL/SQL procedure successfully completed.
```

从输出可以看出，4 种数据类型（实质是三种）对应的变量默认值都为 NULL。为了更清楚地理解这个结果，我们改写上述代码，如实例 5-29 所示。

实例 5-29　改写上例代码。

```
SQL> set serveroutput on;
 declare
  first_name        varchar2(20);
  hire_date   date;
  id          number;
  last_name   employees.last_name%type;
begin
 if first_name is null
   then
      dbms_output.put_line('first_name is null');
 end if;
 if hire_date is null
   then
      dbms_output.put_line('hire_date is null');
 end if;
 if id is null
   then
      dbms_output.put_line('id is null');
 end if;
 if last_name is null
   then
      dbms_output.put_line('last_name is null');
 end if;
end;
.
/
```

我们加入了一个控制语句：if then end if，这样就可以更清楚地判定变量是否为空值，下面我们执行该语句：

```
first_name is null
hire_date is null
id is null
last_name is null

PL/SQL procedure successfully completed.
```

这里我们没有使用常量来定义变量，如果使用常量，则变量的值在整个生命周期中都保持不变，即在变量初始化时定义的值。

5.3.2　变量的有效范围

前面已经介绍了变量，变量在使用前必须声明，可以采用默认值、赋予初始值以及使用常量定义。但是这些变量都有作用范围，离开了作用范围，变量就失效了。若想掌握和使用变量，就必须理解其作用范围。

变量的作用范围就是变量的有效范围，即可以访问该变量的一段程序。一般情况下，变量的作用范围是声明该变量的语句块。这个语句块就是 BEGIN…END 之间的部分。PL/SQL 同时支持使用标签来改进语句的可读性，使得嵌套语句中具有相同名称的变量不会互相混淆。

这个标签可以位于 BEGIN 或者 DELCARE 之前，代码如下所示。

```
set serveroutput on;
 <<outer_part>>
 begin
    dbms_output.put_line('this is a test ');
 end outer_part
```

此时，在 END 之后的 outer_part 表明标签的结束。而嵌套语句就是嵌套在其他语句块中的语句，使用嵌套语句可以限制变量的可见性，如处于嵌套语句中的变量只能在自身语句块中可见，而外层语句块的变量只对自己可见，嵌套语句无法使用。下面我们通过实例 5-30 演示嵌套语句以及涉及的变量可见性问题。

实例 5-30　应用嵌套语句。

```
set serveroutput on;
<<outer_part>>
 declare
  first_name varchar2(20) :='outer_part';
  begin
     <<inner_part>>
     declare
        first_name varchar2(20) :='inner_part';
          begin
             dbms_output.put_line('first_name is : '||first_name);
          end inner_part;
   dbms_output.put_line('first_name is : '||first_name);
 end outer_part;
.
/
```

该语句块包含嵌套语句，嵌套语句使用标签<<inner_part>>说明，我们在嵌套语句内部，以及外部语句块中同时声明了变量 first_name，在外部语句块中为变量 first_name 赋予初始值 outer_part，在嵌套语句块中为变量 first_name 赋予初始值 inner_part，我们分别在嵌套语句块和外部语句块中使用 dbms_output 包打印变量值，从而分析变量的可见性。

执行 PL/SQL 过程，通过实际执行过程来验证变量的可见性问题。

```
SQL> /
first_name is : inner_part
first_name is : outer_part

PL/SQL procedure successfully completed.
```

显然第 1 个输出对应的语句是以下语句块部分。

```
<<inner_part>>
    declare
      first_name varchar2(20) :='inner_part';
       begin
         dbms_output.put_line('first_name is : '||first_name);
       end inner_part;
```

该语句块为内部语句块，输出变量 first_name 的初始值为 inner_part，是我们在嵌套语句中声明的变量。在外部语句块中定义的参数 first_name 的初始值为 outer_part，此时嵌套语句块无法看到外部语句块的变量。

我们再分析第 2 个输出，第 2 个输出对应外部语句块，如实例 5-31 所示。

实例 5-31　外部语句块。

```
set serveroutput on;
<<outer_part>>
 declare
  first_name varchar2(20) :='outer_part';
  begin
    <<inner_part>>
   ……………..
     end inner_part;
   dbms_output.put_line('first_name is : '||first_name);
 end outer_part;
.
/
```

此时输出参数 first_name 的初始值为 outer_part，该值是在外部语句块中定义的，此时嵌套语句块内部的变量 first_name 在外部语句中无法看到，所以此时输出 outer_part，而不是在嵌套语句中赋予的初始值 inner_part。

5.3.3　变量的赋值

变量在使用前必须声明，但是变量初始化并不是必须的。对于变量的初始化，我们可以在声明时为变量赋值，也可以在 PL/SQL 语句块中使用 SELECT INTO 为变量赋值，使用 SELECT INTO 为变量赋值的语法格式如下所示。

```
SELECT 值或者表达式的值 INTO 变量 FROM 表名
```

下面我们使用 SELECT INTO 为变量赋值，如实例 5-32 所示。

实例 5-32　使用 SELECT INTO 为变量赋值。

```
SQL> set serveroutput on;
 declare
  average_sal number(8,2);
 begin
  dbms_output.put_line ('average_sal is : '|| average_sal);
                                              --使用 SELECT INTO 赋值之前
  select  avg(salary) into average_sal from employees;
  dbms_output.put_line ('average_sal is : '|| average_sal);
                                              --使用 SELECT INTO 赋值之后。
```

```
 end;
.
/
```

在上例中，我们定义了一个变量 average_sal，其变量类型为 number(8,2)，用于存储在 PL/SQL 语句块中计算的平均工资数，在语句块的执行部分，我们首先输出了变量 average_sal 的值，此时没有为该变量赋值，接下来我们使用 SELECT INTO 语句为变量赋值，在该语句中我们使用了函数 avg 来计算表 employees 中所有员工的平均工资，并将计算后的值赋予变量 average_sal，最后我们输出该值。执行结果如下所示。

```
SQL>
average_sal is :
average_sal is : 6461.83

PL/SQL procedure
```

从执行结果可以知道，在没有使用 SELECT INTO 语句为变量 average_sal 赋值之前，该变量的值使用默认值，默认值为 NULL。在使用 SELECT INTO 语句为该变量赋值之后，我们再次打印该变量的值，发现赋值成功。计算后的平均工资为 6461.83。

那么，如果通过一个表达式为变量赋值，应该如何操作呢？我们不使用实际表的操作来计算数值，此时可以使用 DUAL 虚表。如下几个实例的目的是更新表 employees 中 FIRST_NAME 为 Alexander 的用户的工资。如实例 5-33 所示，我们先查询该用户的工资。

实例 5-33　查询该用户的工资。

```
SQL> select salary,first_name,last_name from employees
  2 where first_name= 'Alexander';

    SALARY FIRST_NAME           LAST_NAME
---------- -------------------- -------------------------
      9000 Alexander            Hunold
      3100 Alexander            Khoo
```

下面我们使用 PL/SQL 程序来更新该用户的工资，如实例 5-34 所示。

实例 5-34　更新该用户的工资。

```
SQL> declare
  2   var_first_name employees.first_name%type;
  3  begin
  4   select 'Alexander'
  5    into var_first_name
  6    from dual;
  7   update employees
  8    set salary=3100
  9   where first_name='Alexander';
 10* end;
SQL> /

PL/SQL procedure successfully completed.
```

PL/SQL 语句块的执行过程为：首先我们声明了变量 var_first_name，使用 Anchored 变量类型，与表 employees 的列 FIRST_NAME 一致。在语句块的执行部分使用 SELECT INTO 语句为变量

var_first_name 赋值，此时我们直接将一个字符串赋予变量，使用了虚表 DUAL 实现。然后，使用 DML 语句 UPDATE 来更新表 employees 中 first_name 为 Alexander 的用户工资，统一设置为 3100。最后该 PL/SQL 语句块执行成功。下面我们通过查询再次确认是否修改成功，如实例 5-35 所示。

实例 5-35　再次确认是否修改成功。

```
SQL>select salary,first_name,last_name from employees where first_name='Alexander';

    SALARY FIRST_NAME           LAST_NAME
---------- -------------------- -------------------------
      3100 Alexander            Hunold
      3100 Alexander            Khoo
```

显然，从输出结果可以知道 PL/SQL 语句块执行成功。



使用 PL/SQL 语句块执行的任务为一个完整事务，执行完毕后自行提交，更改永久有效。

5.4　序列号

Oracle 的序列是一种数据库对象，用于产生唯一的数字，该数字以一定的间隔增加，序列号对于某些订单系统很有用处。

5.4.1　序列号的定义和特点

Oracle 使用序列生成器自动产生用户可以在事务中使用的唯一序列号，该序列号是一个整数类型数据，序列生成器主要完成在多用户环境下产生唯一的数字序列，但是不会造成额外的磁盘 I/O 或事务锁。简单地说序列号是 Oracle 数据库的一个对象，该对象产生唯一序列号。

在不使用序列号时，如果多个用户同时向 EMP 表中插入一条员工记录，用户必须等待以得到下一个可用的员工号，而一旦使用序列号，则用户无须相互等待就可以得到下一个可用的员工号。序列生成器会自动为每个用户创建正确的员工编号。

由此可见使用序列生成器，能够避免多用户互相等待而造成的事务串行执行，序列号的使用提高了系统的事务处理能力，减少了多用户并行操作的等待时间。

Oracle 的序列号具有如下特点：

- 序列号是独立于表的对象，由 Oracle 自动维护。
- 序列号可以被多个用户共享使用，即同一个序列对象可供多个表使用且互相独立。

5.4.2　序列号的创建和使用

本小节我们将介绍如何创建序列号，以及在 PL/SQL 语句块中使用序列号，在创建序列号之后，就可以调用序列号对象 CURRENT 计算当前序列的值，调用序列号对象的 nextval 递增一个值并返回。下面我们创建一个序列号，同时创建一个表，为使用序列号做准备，如实例 5-36 所示。

实例 5-36　创建一个序列号。

```
SQL> create sequence stuseq start with 2000 increment by 1;

Sequence created.
```

我们创建了一个序列号 stuseq，这个序列号的开始值为 2000，以后每次使用便递增 1 个数字，序列号不会减少。下面我们查询当前序列号的值，如实例 5-37 所示。

实例 5-37　查询当前序列号的值。

```
SQL> select stuseq.currval from dual;
select stuseq.currval from dual
                           *
ERROR at line 1:
ORA-08002: sequence STUSEQ.CURRVAL is not yet defined in this session
```

当我们第一次查询该序列号的值 curval 时，会提示没有定义，我们需要先查询一次 nextval 值，如实例 5-38 所示。

实例 5-38　查询 nextval 值。

```
SQL> select stuseq.nextval from dual;

   NEXTVAL
----------
      2000
```

此时的输出说明，当刚刚创建序列号时，第一次查询的 nextval 值是当前序列号的 start with 对应的值。再次查询 curval 的值，才会出现在序列号定义中与 start with 对应的值，如实例 5-39 所示。

实例 5-39　再次查询当前序列号的值。

```
SQL> select stuseq.currval from dual;

   CURRVAL
----------
      2000
```

到目前为止，我们已创建了一个序列号 stuseq，起始值为 2000，步进值为 1，也就是说每次调用序列号的 nextval，该序列号的值都增加 1。

下面学习如何使用序列号，重建序列号 stuseq，然后创建表 student，如实例 5-40、实例 5-41 所示。

实例 5-40　重建序列号 stuseq。

```
SQL> drop sequence stuseq;

Sequence dropped.
SQL> create sequence stuseq
  2  start with 2000
  3  increment by 1;

Sequence created.
```

实例 5-41　创建表 student。

```
SQL>create table student(student_id number,name varchar2(20),sex char(2), birth_place
varchar2(20));
```

连续使用序列号向表中插入数据。

为了更清楚地查看该表的结构，我们使用 DESC 指令进行查询，如实例 5-42 所示。

实例 5-42　使用 DESC 指令进行查询。

```
SQL> desc student;
 Name                                        Null?     Type
 ------------------------ -----------------------------------
STUDENT_ID                                             NUMBER
 NAME                                                  VARCHAR2(20)
 SEX                                                   CHAR(2)
 BIRTH_PLACE                                       VARCHAR2(20)
```

使用序列号插入该表的列 STUDENT_ID 中，下面向表中插入一行数据，如实例 5-43 所示。

实例 5-43　向表中插入一行数据。

```
SQL>  insert into student values(
  2  stuseq.nextval,'tom','M','Boston');

1 row created.

SQL> commit;

Commit complete.
```

下面我们查询表 student 中的数据，以确认插入效果，如实例 5-44 所示。

实例 5-44　查询表 student 中的数据。

```
SQL> insert into student values(stuseq.nextval,'tom','F','Boston');

1 row created.

SQL> select * from student;

STUDENT_ID NAME                 SE BIRTH_PLACE
---------- -------------------- -- --------------------
     2000 tom                  F  Boston
```

可以看到，当我们第一次调用序列号 stuseq.nextval 时，返回的值为定义序列号的开始值 2000，以后每次调用 stuseq.nextval 都会返回一个递增的值，步进值为 1，如实例 5-45 所示。

实例 5-45　再次插入数据。

```
SQL>  insert into student values(stuseq.nextval,'Kyte','F','Newyork');

1 row created.
```

此时，我们向表 student 中再次插入一行数据，这是第 2 次调用 stuseq.nextval 了，此时的 STUDENT_ID 应该为 2001。下面我们查询插入结果，如实例 5-46 所示。

实例 5-46　查询插入结果。

```
SQL> select * from student;

STUDENT_ID NAME                 SE BIRTH_PLACE
---------- -------------------- -- --------------------
      2000 tom                  F  Boston
      2001 Kyte                 F  Newyork
```

5.5　事务

事务具有一个很重要的特性——原子性，即要求一个事务要么完成，要么不做。通俗地讲就是“要么不做，做就做完”，所以也就引出一个问题，如果一个长 PL/SQL 语句块中间涉及很多 DML 操作，若在最后一个 DML 操作完成前发生异常，我们希望将 PL/SQL 语句块回退到前面合适的某个位置，当然我们先假设这个 PL/SQL 语句是一个完整的事务（实际上，一个 PL/SQL 块可以包含多个事务）。使用 SAVEPOINT 就可以很好地解决这一点，通过保留点技术插入 PL/SQL 语句块的合理位置，这样就可以更好地控制事物的粒度，从而将事务分解为更小的单元，更好地控制整个事务的执行。

为了控制整个事务，我们需要三个关键字显式地控制事务的执行，即 COMMIT、ROLLBACK 和 SAVEPOINT。下面我们分别介绍它们的作用，并通过实例进行演示。

5.5.1　COMMIT

提交的作用是保证数据更改永久有效，即将更改的数据保存到物理数据文件或者已经写入重做日志并标识为提交。下面我们更新表 employees 中 employee_id=206 的员工数据，修改薪水为 9000。先查询该用户的信息，如实例 5-47 所示。

实例 5-47　查询该用户的信息。

```
SQL>select first_name,last_name,hire_date,salary,department_id from employees
  2* where employee_id=206

FIRST_NAME       LAST_NAME    HIRE_DATE      SALARY  DEPARTMENT_ID
-------- ---- ------ ----------- ---------------------------
William             Gietz           07-JUN-02       8300          110
```

在修改前，该用户 William 的薪水为 8300。下面我们执行一个更新操作，如实例 5-48 所示。

实例 5-48　执行更新操作。

```
SQL>  update employees set salary=9000 where employee_id=206;

1 row updated.
```

此时，我们修改了用户的数据，但是没有提交，即没有执行 COMMIT 操作，下面查询该用户的信息，以便确认数据 SALARY 是否修改，如实例 5-49 所示。

实例 5-49　查询更改是否成功。

```
SQL>select first_name,last_name,hire_date,salary,department_id from employees
```

```
  2* where employee_id=206

FIRST_NAME       LAST_NAME    HIRE_DATE     SALARY DEPARTMENT_ID
-------- ---- ------ ----------- ---------------------------
William            Gietz          07-JUN-02      9000         110
```

从输出可以看出更改后的效果，那么如果重新登录是否还能看到这个结果呢？如实例 5-50 所示。

实例 5-50　重新登录后查询。

```
[oracle@localhost ~]$ sqlplus hr/oracle

SQL*Plus: Release 11.2.0.1.0 Production on Wed Dec 14 10:23:10 2011

Copyright (c) 1982, 2009, Oracle.  All rights reserved.


Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> select first_name,last_name,hire_date,salary,department_id from employees
  2* where employee_id=206

FIRST_NAME       LAST_NAME    HIRE_DATE     SALARY DEPARTMENT_ID
-------- ---- ------ ----------- ---------------------------
William            Gietz          07-JUN-02      8300          110
```

从输出可以看出，当我们打开一个新的会话，使用相同的用户名登录时，此时无法看到上一个会话修改的结果，原因就是没有提交，所以 Oracle 必须保证数据的一致性，即数据更新被提交前必须保证新用户查询的是该数据的原始数据，而不是更新后的数据。下面我们返回到上一个更新会话，执行提交操作，如实例 5-51 所示。

实例 5-51　执行提交操作。

```
SQL> commit;

Commit complete.
```

提交意味着数据永久保存到数据库，其他任何用户查询该表中用户 William 的数据时都会看到一致的修改后的数据，如实例 5-52 所示。

实例 5-52　重新登录后查询。

```
[oracle@localhost ~]$ sqlplus hr/oracle

SQL*Plus: Release 11.2.0.1.0 Production on Wed Dec 14 10:28:41 2011

Copyright (c) 1982, 2009, Oracle.  All rights reserved.


Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> set line 120
SQL> select first_name,last_name,hire_date,salary,department_id from employees
```

```
  2* where employee_id=206

FIRST_NAME       LAST_NAME    HIRE_DATE     SALARY DEPARTMENT_ID
-------- ---- ------ ----------- ---------------------------
William            Gietz          07-JUN-02       9000          110
```

通过上面的实例，读者可以清楚地理解 COMMIT 的作用，下面我们介绍 ROLLBACK 的作用。

5.5.2　ROLLBACK

回滚意味着将事务的操作取消，即执行和事务相反的操作，将数据恢复到原始状态，下面我们通过实例 5-53 说明这个问题。同样修改 employee_id=206 的用户的 salary 为 9000，更新之后不提交数据，而是回滚事务，即取消数据修改，将数据恢复到原始数值。

实例 5-53　更新数据。

```
SQL>  update employees set salary=9000 where employee_id=206;

1 row updated.
```

接下来查询这个修改，此时在当前会话中可以看到数据被更新，如实例 5-54 所示。

实例 5-54　查询修改是否成功。

```
SQL>select first_name,last_name,hire_date,salary,department_id from employees
  2* where employee_id=206

FIRST_NAME       LAST_NAME    HIRE_DATE     SALARY DEPARTMENT_ID
-------- ---- ------ ----------- ---------------------------
William            Gietz          07-JUN-02       9000          110
```

注意，此时其他用户不会看到这个修改，因为没有提交，事务不会结束。下面我们回滚该事务，继续查询该数据是否恢复到初始值，如实例 5-55 所示。

实例 5-55　回滚事务。

```
SQL> rollback;

Rollback complete.
```

回滚后，Oracle 将使用 UNDO 中记录的数据，回滚 salary 到其初始值 8300，如实例 5-56 所示。

实例 5-56　查询数据。

```
SQL>select first_name,last_name,hire_date,salary,department_id from employees
  2  where employee_id=206;

FIRST_NAME  LAST_NAME      HIRE_DATE     SALARY DEPARTMENT_ID
-- -------- -- -------------- -------------------------
William        Gietz              07-JUN-02        8300          110
```

5.5.3　SAVEPOINT

保留点可以将大的 PL/SQL 块分解，使得用户可以更灵活地控制 PL/SQL 语句，下面我们设计

一个 PL/SQL 程序，从而体会使用 SAVEPOINT 的作用，如实例 5-57 所示。

实例 5-57　使用 SAVEPOINT。

```
SQL> declare
  2    var_salary number(8,2);
  3  begin
  4     update employees set salary=9000 where employee_id=206;
  5     select salary into var_salary from employees where employee_id=206;
  6        dbms_output.put_line (' the first step salary is : '||var_salary);
  7     savepoint a1;
  8     update employees set salary=10000 where employee_id=206;
  9     select salary into var_salary from employees where employee_id=206;
 10        dbms_output.put_line (' the second step salary is : '||var_salary);
 11     savepoint a2;
 12    update employees set salary=11000 where employee_id=206;
 13    select salary into var_salary from employees  where employee_id=206;
 14        dbms_output.put_line (' the third step salary is : '||var_salary);
 15    savepoint a3;
 16   rollbackup to a1;
 17* end;
```

字体加粗的部分为定义的三个保留点，在保留点 a1 之前，我们更改 employee_id=206 的用户 salary 为 9000，在保留点 a2 之前，我们更改 employee_id=206 的用户 salary 为 10000，注意，如果此时提交数据，之前的 9000 将被永久覆盖，但是此时事务没有结束，我们继续更改 employee_id=206 的用户 salary 为 11000，此时再设置一个保留点，最后我们将整个 PL/SQL 程序恢复到保留点 a1，这样，保留点 a1 之前的修改将得以保留，而之后的修改都将回滚。下面将执行这个 PL/SQL 程序，输出如下所示。

```
the first step salary is : 9000
the second step salary is : 10000
the third step salary is : 11000

PL/SQL procedure successfully completed.
```

从输出可以知道在执行 rollback to a1 之前，更新操作都已执行成功，但是在 PL/SQL 程序的最后，我们使用了 rollback to a1 将整个 PL/SQL 语句恢复到保留点 a1 之前的操作结果，而保留点 a1 之后的部分都回滚。当整个 PL/SQL 语句执行完毕之后，查询修改结果，如实例 5-58 所示。

实例 5-58　查询修改结果。

```
SQL> select salary from employees where employee_id=206;

    SALARY
----------
      9000
```

从输出可以知道，这个输出结果是保留点 a1 之前的数据更新结果。显然，保留点 a1 之后的部分都忽略了。

有个问题需要读者注意，下面更改 PL/SQL 语句块，修改部分用粗体字显示，如实例 5-59 所示。

实例 5-59　更改 PL/SQL 语句块。

```
SQL> declare
```

```
  2   var_salary number(8,2);
  3  begin
  4    update employees set salary=9000 where employee_id=206;
  5    select salary into var_salary from employees where employee_id=206;
  6      dbms_output.put_line ('the first step salary is : '||var_salary);
  7    savepoint a1;
  8    update employees set salary=10000 where employee_id=206;
  9    select salary into var_salary from employees where employee_id=206;
 10      dbms_output.put_line ('the second step salary is : '||var_salary);
 11    savepoint a2;
 12   update employees set salary=11000 where employee_id=206;
 13   select salary into var_salary from employees  where employee_id=206;
 14      dbms_output.put_line ('the third step salary is : '||var_salary);
 15   savepoint a3;
 16  rollback to a1;
 17      dbms_output.put_line ('the final step salary is : '||var_salary);
 18* end;
```

在保留点 a1 语句执行完毕之后，打印 var_salary 的最终值，读者可以考虑此时的输出结果，是 9000 还是 11000 呢？我们给出执行结果，再解释这个问题。

```
the first step salary is : 9000
the second step salary is : 10000
the third step salary is : 11000
the final step salary is : 11000

PL/SQL procedure successfully completed.
```

我们看到在 PL/SQL 语句块执行了 rollback to a1 之后，变量 var_salary 的值是 11000，而不是保留点 a1 之前的数据更新。这个问题就涉及到在执行完 rollback to a1 之后 Oracle 到底做了些什么。现总结如下。

- PL/SQL 语句块还没有完全结束，所以事务也没有结束，在紧随 rollback to savepoint 之后看到的数据与保留点无关。
- 释放保留点之后的所有 SQL 语句所拥有的锁和资源。
- 取消保留点之后的所有操作，但是保留点依然是活跃的，直到执行提交 COMMIT 或者 ROLLBACK，所以我们看到的依然是第 3 个更新语句修改的 salary 结果。

5.6　本章小结

本章是 PL/SQL 编程的基础部分，这部分主要包括了 PL/SQL 的数据类型，这些类型包括常用的 CHAR、VARCHAR2、NUMBER、LONG、LONG RAW、BOOLEAN、PLS_INTEGER、DATE、TIMESTAMP、ANCHORED 以及用户自定义的数据类型，然后介绍了 PL/SQL 中的保留字，保留字是 PL/SQL 独有的部分，程序员不能使用。在变量定义和初始化部分介绍了如何定义变量和初始化变量，最后介绍了序列号的使用并且从本质上解释了 COMMIT、ROLLBACK 和 SAVEPOINT 的含义，这些对于编写 PL/SQL 程序都十分重要。

第 6 章

PL/SQL程序流程

在任何编程语言中，都需要一种控制程序流程的语句。因为任何程序都不是顺序执行的，必然会使用控制语句来控制程序的执行流程，通过各种判断来改变流程走向，从而完成各种复杂的业务需求。PL/SQL 是一种过程控制语言，必然需要一种控制过程的方式。本章我们将介绍控制 PL/SQL 程序流程的语句。

6.1 IF语句

IF 语句是一种条件判断语句，这种语句首先执行一个条件判断，如果满足，则执行接下来的语句，否则，根据情况继续执行。IF 条件语句具有两种形式：一种是 IF-THEN 形式；另一种是 IF-THEN-ELSE 形式。下面我们分别介绍这两种形式的 IF 语句，以及 ELSIF 语句、嵌套 IF 语句。

6.1.1 IF-THEN 语句

IF-THEN 是最简单明了的条件语句，正如我们日常生活中的判断，“如果周末天气好，就去爬香山”，显然这是一个判断，判断的前提为 IF 部分的内容，即“周末天气好”，则 THEN 执行的结果是“去爬香山”。在我们日常生活中这种实例比比皆是，在程序流程控制中，IF-THEN 语句也是很常见的一种控制语句。其语法结构如下所示。

```
IF 条件1  THEN
执行语句1
……
执行语句n
END  IF;
```

在上述语法结构中紧随 IF 之后的部分是执行条件，一旦条件满足，则执行 THEN 之后的语句，从执行语句 1 到执行语句 n 依次执行，直至 END IF 语句的结束部分。如果 IF 中的条件不满足，则跳转到 END IF 之后的程序段部分。IF-THEN 语句的执行流程图如图 6-1 所示。

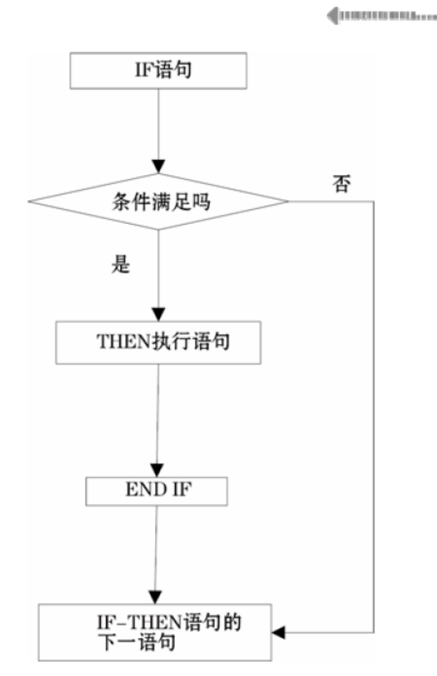


图 6-1　IF-THEN 语句流程图

下面我们通过实例 6-1 进一步理解 IF-THEN 语句的用法。

实例 6-1　应用 IF-THEN 语句。

```
SQL> set serveroutput on;
SQL> declare
  2   var_num  number;
  3   var_first_name employees.first_name%type := '&first_name';
  4  begin
  5    select count(*) into var_num from employees where first_name=var_first_name;
  6    if var_num>=1 then
  7      dbms_output.put_line('there exists :'||var_first_name);
  8    end if;
  9* end;
```

在上面的 PL/SQL 语句块中使用了 IF-THEN 语句：首先定义了两个变量，其中一个是 var_num，另一个是 var_first_name，第 2 个变量用于控制员工姓氏的输入，第 1 个变量用于存储一个数字，如果该姓的员工存在，则将该姓的员工数量赋予该变量，显然如果该变量的值大于等于 1，则说明该姓的员工存在，并输出提示。这是一个典型的 IF-THEN 语句。

下面执行该过程：

```
Enter value for first_name: Alexander
old   3:  var_first_name employees.first_name%type := '&first_name';
new   3:  var_first_name employees.first_name%type := 'Alexander';
there exists :Alexander

PL/SQL procedure successfully completed.
```

我们对替代变量赋值 Alexander，此时语句块执行，输出结果说明在 IF 条件中值 var_num>=1，这个条件满足，所以执行了 THEN 之后的部分语句。

继续执行该语句，输入一个在表 employees 中不存在的姓，从而理解 IF-THEN 语句的缺陷，如实例 6-2 所示。

实例 6-2　输入一个在表 employees 中不存在的姓。

```
SQL>declare
  2   var_num  number;
  3   var_first_name employees.first_name%type := '&first_name';
  4  begin
  5  select count(*) into var_num from employees where first_name=var_first_name;
  6    if var_num>=1 then
  7      dbms_output.put_line('there exists :'||var_first_name);
  8    end if;
  9* end;
SQL> /
Enter value for first_name: linshuze
old   3:  var_first_name employees.first_name%type := '&first_name';
new   3:  var_first_name employees.first_name%type := 'linshuze';

PL/SQL procedure successfully completed.
```

我们发现，姓 linshuze 在表中不存在，所以没有任何输出，这显然不是我们希望看到的，因为对于用户来讲，更希望看到一个结果，即有还是没有，而不是保持“沉默”的状态，这样的程序设计对用户而言是不友好的。为了解决这个问题下面再来学习 IF-THEN-ELSE 语句。

6.1.2　IF-THEN-ELSE 语句

该语句也是条件判断语句，语句的语法格式如下所示。

```
IF 条件 1   THEN
   执行语句 1;
   ......
   执行语句 n;
  ELSE
  执行语句 1;
   ......
   执行语句 n;
END IF;
```

下面分析这个语句的执行流程：首先执行 IF 语句，如果条件满足，则执行 THEN 之后的语句，当 THEN 之后的语句执行完毕，程序跳转到 END IF，结束语句的执行。如果 IF 语句的条件不满足，则执行 ELSE 之后的语句，ELSE 之后的语句执行完毕后，再执行 END IF，从而结束语句执行。IF-THEN-ELSE 语句的流程图如图 6-2 所示。

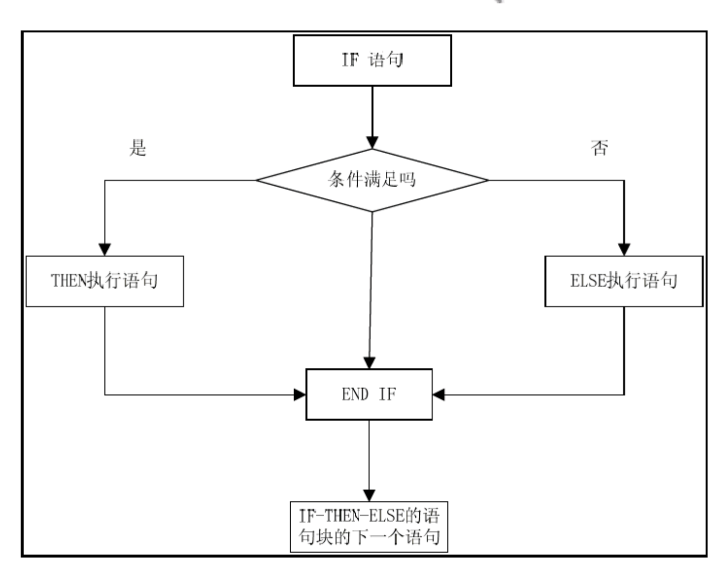


图 6-2　IF-THEN-ELSE 流程图

下面我们更改上节的实例代码，添加 ELSE 执行部分，即告诉程序，如果 IF 条件不满足，不是转到 END IF 后的语句继续执行，而是执行 ELSE 语句之后的部分。我们用粗体字显示出修改部分，如实例 6-3 所示。

实例 6-3　更改上节的实例代码。

```
SQL> declare
  2   var_num  number;
  3   var_first_name employees.first_name%type := '&first_name';
  4  begin
  5   select count(*) into var_num from employees where first_name=var_first_name;
  6    if var_num>=1 then
  7     dbms_output.put_line('there exists :'||var_first_name);
  8     else
  9     dbms_output.put_line('the '||var_first_name||' does not exist!');
 10    end if;
 11* end;
SQL> /
```

在上例中，我们使用了 IF-THEN-ELSE 语句。通过 ELSE 语句，将不满足 IF 条件时的语句流程跳转到 ELSE 语句部分，此时更加灵活地控制了程序的执行流程。当我们输入的姓不存在时，就不会出现“沉默”的情况，而是提示该用户不存在，执行该过程，效果如下所示。

```
Enter value for first_name: linshuze
old   3:  var_first_name employees.first_name%type := '&first_name';
new   3:  var_first_name employees.first_name%type := 'linshuze';
the linshuze does not exist!

PL/SQL procedure successfully completed.
```

在执行该过程时，我们输入了姓 linshuze，由于该用户不存在，所以程序的 IF 条件 var_num>=1 不满足，程序就会跳转到 ELSE 之后的语句。最后执行 END IF，结束整个 IF-THEN-ELSE 语句过程。

6.1.3 ELSIF 语句

在复杂的条件程序流程中，可能存在多个条件语句，并且这些条件是互斥的关系，此时就可以使用 ELSIF 语句。ELSIF 语句的语法如下所示。

```
IF 条件 1
THEN
   执行语句 1;
ELSIF 条件 2
   执行语句 2;
ELSIF 条件 3
   执行语句 3;
  ……
ELSIF 条件 n
    执行语句 n;
ELSE
执行语句;
END IF;
```

程序的执行流程是：依次判断 IF、ELSIF 的条件，只要一个满足条件，则执行条件之后的语句，然后跳转到程序结束的 END IF 部分。如果所有条件都不满足，则跳转到 ELSE 部分，执行 ELSE 后的语句，最后跳转到 END IF 结束语句。下面我们编写实例 6-4 来演示该语句的应用。

实例 6-4　应用 ELSIF 语句。

```
SQL> create or replace PROCEDURE p (sales NUMBER)
IS
bonus NUMBER := 0;
BEGIN
IF sales > 50000 THEN
bonus := 1500;
ELSIF sales > 35000 THEN
bonus := 500;
ELSE
bonus := 100;
END IF;
DBMS_OUTPUT.PUT_LINE (
'Sales = ' || sales || ', bonus = ' || bonus || '.'
);
END p
  /

Procedure created.
```

在上例中，我们创建了一个存储过程。该过程允许输入一个销售额，根据销售额给予奖金：如果销售额大于 50000，则奖金为 1500；如果销售额大于 35000，则奖金为 500；其他情况奖金为 100。注意因为条件是互斥的，并且 ELSIF 的执行过程是顺序的，所以必须将条件中销售量大的判断放在前面。

下面查询该过程的参数信息，如实例 6-5 所示。

实例 6-5　查询该过程的参数信息。

```
SQL> desc p;
PROCEDURE p
```

```
  Argument Name                  Type                In/Out Default?
  ----------- ---- ----------------------------------------------
  SALES                        NUMBER                IN
```

如果想知道当前用户具有多少过程，可以通过 user_objects 数据字典查询，只要在查询谓词中限制查询类型为 PROCEDURE 即可。

下面我们调用该过程，代码如下所示。

```
SQL> BEGIN
  2  P(50001);
  3  P(37000);
  4  P(1000);
  5  END;
  6  /
Sales = 50001, bonus = 1500.
Sales = 37000, bonus = 500.
Sales = 1000, bonus = 100.

PL/SQL procedure successfully completed.
```

下面分析一下执行结果：第 1 个调用输入的参数是 50001，此时满足第 1 个 IF 条件语句，所以此时就执行该语句之后的语句，增加的 BONUS 为 1500，因为第 1 个条件满足，所以程序不再继续执行，而是跳转到 END IF 之后，结束当前的语句块；第 2 个调用输入的参数为 37000，程序会首先比较第 1 个 IF 语句，发现条件不匹配，则执行 ELSIF 语句，发现此时条件满足，则执行 ELSIF 语句之后的语句，增加的 BONUS 为 500；第 3 个输入参数为 1000，此时依次比较 IF、ELSIF 语句，发现都没有满足条件的语句，则跳转到 ELSE 语句，执行 ELSE 语句之后的语句，一旦执行语句结束，则跳转到 END IF 语句。



ELSIF 语句是顺序执行的，各个执行条件之间是互斥的关系，在使用该语句时需要注意条件的顺序关系。

6.1.4　嵌套 IF 语句

嵌套 IF 语句是在 PL/SQL 语句块的条件语句中，或其他语句中进一步进行条件限制，将 IF 语句嵌套进去。通过下面的实例 6-6 演示该语句的用法。

实例 6-6　嵌套 IF 语句。

```
SQL>declare
  2   var_num  number;
  3   var_first_name employees.first_name%type := '&first_name';
  4  begin
  5   select count(*) into var_num from employees where first_name=var_first_ name;
  6   if var_num>=1 then
  7     dbms_output.put_line('there exists :'||var_first_name);
  8         if var_first_name='Stephen' then
  9           dbms_output.put_line('Stephen is found');
 10         else
 11           dbms_output.put_line('where is Stephen?');
 12         end if;
 13     else
```

```
 14     dbms_output.put_line('the '||var_first_name||' does not exist!');
 15    end if;
 16* end;
```

上例中的粗体字部分是我们加入的嵌套 IF 语句，即如果我们输入的某个姓存在，则进行进一步判断，即该姓是否为 Stephen，如果是，则输出发现该姓氏，否则输出一个疑问'where is Stephen?'。然后结束嵌套的 IF 语句，进入外层嵌套的 END IF 语句结束外层 IF 判断条件。

下面我们执行调用该过程。

```
Enter value for first_name: Stephen
old   3:  var_first_name employees.first_name%type := '&first_name';
new   3:  var_first_name employees.first_name%type := 'Stephen';
there exists :Stephen
Stephen is found

PL/SQL procedure successfully completed.
```

在执行该语句时，我们输入了 Stephen，显然由于该姓存在，所以满足外层 IF 条件，执行 THEN 语句部分，此时会输出该姓信息，然后进入嵌套 IF 语句，继续判断该姓是否为 Stephen，如果是，则输出 Stephen is found，然后结束内层嵌套，返回到外层嵌套的 END IF，结束外层循环。

6.2 CASE语句

CASE 语句也是一种条件判断语句，通过选择器来判断条件是否满足，CASE 语句有两种形式：一种是简单的 CASE 语句，另一种是搜索式 CASE 语句。下面我们介绍这两类 CASE 语句。

6.2.1 简单的 CASE 语句

简单的 CASE 语句通过选择器来进一步确定要执行的动作，下面是 CASE 语句的语法。

```
CASE 选择器
   WHEN  选择器的值 1  THEN 执行语句 1
   WHEN  选择器的值 2  THEN 执行语句 2
   ……
   WHEN  选择器的值 n  THEN 执行语句 n
   ELSE
    执行语句;
END  CASE;
```

在简单的 CASE 语句中，可以使用选择器确定进一步的行为，通过不同选择器的值来确定执行语句，在执行时，程序会逐一判断选择器的值，如果匹配，则执行之后的 THEN 语句，否则继续判断，如果都不满足，则跳转到 ELSE 语句，最后结束 CASE 语句。

下面给出一个实例进行说明，如实例 6-7 所示。

实例 6-7　应用 CASE 语句。

```
SQL> declare
 mark number :=&mark_number;
begin
 case mark
```

```
      when 1 then
        dbms_output.put_line('Monday');
      when 2 then
        dbms_output.put_line('Tuesday');
      when 3 then
        dbms_output.put_line('Wednesday');
      when 4 then
        dbms_output.put_line('Thursday');
      when 5 then
        dbms_output.put_line('Friday');
      when 6 then
        dbms_output.put_line('Saturday');
      when 7 then
        dbms_output.put_line('Sunday');
     end case;
    end;
```

在这个实例中选择器是一个数字，选择器的名为 mark，其值为 1~7，每个数字对应一周中的一天，一旦匹配，则打印相应的日期。

下面是该过程的执行结果。

```
Enter value for mark_number: 3
old   2:  mark number :=&mark_number;
new   2:  mark number :=3;
Wednesday

PL/SQL procedure successfully completed.
```

在执行过程中，我们输入了数字 3，此时执行 CASE 语句，开始匹配选择器，将依次执行匹配过程。一旦找到选择器的值为 3 的匹配行，则输出该 WHEN 语句之后的执行语句，即输出 Wednesday。

但是，这个语句有个缺陷：如果用户输入一个值 100，此时会输出什么呢？显然此时没有匹配任何值，也不会有任何输出，所以需要增加一个处理该“异常”的语句，下面我们使用 ELSE 语句来处理这个问题，如实例 6-8 所示。

实例 6-8　添加 ELSE 语句。

```
SQL>declare
  2   mark number :=&mark_number;
  3  begin
  4   case mark
  5    when 1 then
  6      dbms_output.put_line('Monday');
  7    when 2 then
  8      dbms_output.put_line('Tuesday');
  9    when 3 then
 10      dbms_output.put_line('Wedensday');
 11    when 4 then
 12      dbms_output.put_line('Thursday');
 13    when 5 then
 14      dbms_output.put_line('Friday');
 15    when 6 then
 16      dbms_output.put_line('Saturday');
 17    when 7 then
 18      dbms_output.put_line('Sunday');
```

```
 19    else dbms_output.put_line('no such day!');
 20   end case;
 21* end;
```

我们增加了一行 ELSE 语句，该语句在所有 WHEN 语句无法匹配时执行，然后结束 CASE 语句。执行结果如下所示。

```
Enter value for mark_number: 100
old   2:  mark number :=&mark_number;
new   2:  mark number :=100;
no such day!

PL/SQL procedure successfully completed.
```

6.2.2　搜索式 CASE 语句

搜索式 CASE 语句不使用选择器，而是直接使用 WHEN 子句进行直接搜索，将条件放入 WHEN 语句，只要条件满足，则执行对应的 THEN 之后的语句，然后结束该 CASE 语句，如果都没有匹配条件，则结束 CASE 语句或者执行 ELSE 语句，如实例 6-9 所示。

实例 6-9　应用搜索式 CASE 语句。

```
SQL> declare
  2  v_num number :=&var_num;
  3  begin
  4  if v_num >=0 and v_num<101 then
  5   case
  6    when v_num>=90 then
  7      dbms_output.put_line('exellent');
  8    when v_num>=70 then
  9      dbms_output.put_line('good');
 10    when v_num>=60 then
 11      dbms_output.put_line('so bad!');
 12   end case;
 13  else
 14    dbms_output.put_line('no such mark');
 15  end if;
 16* end;
```

在上例中，我们定义了一个 number 类型的变量 v_num，然后在执行语句中使用：首先进行一个判断，以保证我们输入的成绩在 0~100 之间。如果这个条件可以满足，则继续执行 CASE 语句，该 CASE 语句使用搜索方式，即没有使用选择器，而是直接使用条件搜索，一旦满足条件则执行随后的 THEN 子句，从而结束 CASE 语句。如果没有满足条件的 WHEN 语句，则直接回到 ELSE 语句（前提是设计了该语句）或者返回 END IF，结束 CASE 语句。

执行该匿名过程，效果如下所示。

```
Enter value for var_num: 91
old   2: v_num number :=&var_num;
new   2: v_num number :=91;
exellent

PL/SQL procedure successfully completed.
```

我们输入了数字 91，显然它满足第一个 WHEN 搜索语句，所以执行随后的 THEN 语句。如果我们输入 1000，则由于程序在 IF 条件处已经被过滤掉了，根本不会进入到 CASE 语句部分，则直接跳转到 ELSE 处，执行 ELSE 之后的语句，效果如下所示。

```
Enter value for var_num: 1000
old   2: v_num number :=&var_num;
new   2: v_num number :=1000;
no such mark

PL/SQL procedure successfully completed.
```

6.3　循环控制语句

循环控制语句是在一定的周期执行同样的事情，这个周期就是循环次数，可以使用循环来方便地遍历一组数据，例如使用循环来遍历游标，获取从数据库中读到的数据。循环控制语句依据需求的不同分为简单循环、WHILE 循环、FOR 循环。下面我们依次介绍这些循环语句。

6.3.1　简单循环语句

简单循环语句的语法格式如下所示。

```
LOOP
  执行语句
   END LOOP。
```

简单循环从关键字 LOOP 开始，然后是一个执行语句，使用简单循环语句时往往需要一个条件来停止循环，我们使用 EXIT 语句退出循环，如实例 6-10 所示。

实例 6-10　使用简单循环语句。

```
SQL>declare
  2    var_num number :=0;
  3  begin
  4    loop
  5      dbms_output.put_line('var_num is '||to_char(var_num));
  6      var_num := var_num+2;
  7        if var_num>10 then
  8           exit;
  9        end if;
 10     end loop;
 11   dbms_output.put_line('last var_num is '||to_char(var_num));
 12* end;
```

在上例中，定义了一个变量 var_num，赋予初始值为 0，然后开始执行语句块，在 LOOP 循环中，首先输出当前变量 var_num 的值，然后在变量 var_num 当前值的基础上执行一个+2 运算，并进行一个判断，即值 var_num 是否大于 10，如果是，则使用 exit 结束循环，如果不是，则继续执行循环内的语句，输出 var_num 的值，执行+2 计算并进行判断，直到不满足条件 var_num>10 为止。通过程序逻辑我们可以判断 var_num 的输出值将依次是 0、2、4、6、8、10，而最后输出的 last var_num 的值为 12。下面执行该过程，执行结果如下所示。

```
SQL> /
var_num is 0
var_num is 2
var_num is 4
var_num is 6
var_num is 8
var_num is 10
last var_num is 12

PL/SQL procedure successfully completed.
```

在上例中我们使用一个条件进行判断，即使用 EXIT 语句结束了当前的 LOOP 循环，也可以使用 EXIT WHEN 来结束循环，即在 EXIT 的语句后使用 WHEN 执行条件判断，下面我们改写上例，如实例 6-11 所示。

实例 6-11　改写上例代码。

```
SQL> declare
  2    var_num number :=0;
  3  begin
  4    loop
  5      dbms_output.put_line('var_num is '||to_char(var_num));
  6      var_num := var_num+2;
  7        exit when var_num>12;
  8     end loop;
  9   dbms_output.put_line('last var_num is '||to_char(var_num));
 10* end;
```

使用 EXIT WHEN 循环语句时，流程是先执行语句再执行判断，如果满足 WHEN 之后的判断条件则结束循环。在上例中，我们考虑一个临界情况：当 var_num=12 时，执行 EXIT WHEN 语句，显然不满足 var_num>12，所以继续循环，此时输出 var_num 的值为 12，然后执行+2 计算，再次执行 EXIT WHEN 语句，此时 var_num=14，显然该值满足 var_num>12，所以结束循环，当循环结束时，var_num 的值为 14，执行结果如下所示。

```
SQL>
var_num is 0
var_num is 2
var_num is 4
var_num is 6
var_num is 8
var_num is 10
var_num is 12
last var_num is 14

PL/SQL procedure successfully completed.
```

6.3.2　WHILE 循环语句

WHILE 循环的语法结构如下所示。

```
WHILE 循环条件 LOOP
        执行语句 1;
        执行语句 2;
        ……
```

```
        执行语句 n;
        END LOOP;
```

WHILE 循环的使用也非常明了：首先判断循环条件，如果条件满足，则执行循环中的语句，然后结束循环；如果条件不满足，则退出 WHILE 循环。WHILE 循环先判断执行条件，再执行语句。循环条件同时也是结束循环的条件。通过实例 6-12 可以更加具体地讲解 WHILE 循环的用法。

实例 6-12　应用 WHILE 循环。

```
SQL> l
  1  declare
  2    var_num number :=0;
  3  begin
  4   while var_num<12 loop
  5      dbms_output.put_line('var_num is '||to_char(var_num));
  6      var_num := var_num+2;
  7     end loop;
  8   dbms_output.put_line('last var_num is '||to_char(var_num));
  9* end;
```

在上例中，粗体字部分就是 WHILE 循环语句，首先使用 WHILE 的循环条件语句判断循环条件是否成立，循环条件是变量 var_num<12，循环执行语句首先打印当前的 var_num 值，然后执行运算 var_num := var_num+2;在循环体中的语句执行完毕之后会再次回到 WHILE 循环的条件语句，判断循环是继续执行还是退出循环。

下面是该过程的执行结果。

```
SQL> /
var_num is 0
var_num is 2
var_num is 4
var_num is 6
var_num is 8
var_num is 10
last var_num is 12

PL/SQL procedure successfully completed.
```

下面选择一个临界点进行分析：当 WHILE 循环中输出的 var_num 值为 10 时，将执行 var_num := var_num+2 的计算，此时的 var_num 值为 12，然后程序逻辑返回到 WHILE 循环的开始处进行条件判断，发现 var_num<12 不成立（此时 var_num=12），所以，退出 WHILE 循环。执行一条输出语句，即打印当前的 var_num 值。

为了更灵活地控制 WHILE 循环的结束，我们可以在循环体中嵌入 IF 条件语句来提前结束循环，这样我们就可以更灵活地控制程序的流程。下面是典型的语法结构。

```
WHILE 循环条件 LOOP
        执行语句 1;
        执行语句 2;
       IF 结束条件 THEN
          EXIT;
       END IF;
     END LOOP;
```

下面我们给出实例 6-13，通过该实例可以更好地理解如何在循环体中设置条件，从而提前结

束循环的执行。

实例 6-13　在循环体中嵌入 IF 条件语句。

```
SQL> declare
  var_num number :=0;
begin
 while var_num<12 loop
   dbms_output.put_line('var_num is '||to_char(var_num));
      if var_num = 8 then
        exit;
       end if;
   var_num := var_num+2;
  end loop;
 dbms_output.put_line('last var_num is '||to_char(var_num));
end;
.
/
```

在上例中的粗体部分，我们使用一个 IF-THEN 语句来提前结束 WHILE 循环，提前结束循环的条件是 var_num=8，显然，当循环体输出 var_num 的值为 8 时，接下来会直接执行一个 IF 判断，因为满足条件，所以结束循环，程序直接跳转到 END LOOP 后的第一条语句，此时 var_num 的值不会被执行+2 计算，因此最后 var_num 的输出结果是 8，而不是 10。下面我们执行该过程，执行结果如下所示。

```
SQL>
var_num is 0
var_num is 2
var_num is 4
var_num is 6
var_num is 8
last var_num is 8

PL/SQL procedure successfully completed.
```

这里我们使用了 IF-THEN 语句提前结束循环，同样，我们也可以使用 EXIT WHEN 语句结束循环，如实例 6-14 所示。

实例 6-14　使用 EXIT WHEN 语句结束循环。

```
SQL> l
  1  declare
  2    var_num number :=0;
  3  begin
  4   while var_num<12 loop
  5      dbms_output.put_line('var_num is '||to_char(var_num));
  6        exit when var_num=8;
  7      var_num := var_num+2;
  8     end loop;
  9   dbms_output.put_line('last var_num is '||to_char(var_num));
 10* end;
```

粗体部分是用于提前结束循环的 EXIT WHEN 语句，此时的条件也是 var_num=8，读者只要执行一次循环分析，就会知道这个过程与使用 IF-THEN 提前结束 WHILE 循环的语句执行结果一致。下面是执行结果。

```
SQL> /
var_num is 0
var_num is 2
var_num is 4
var_num is 6
var_num is 8
last var_num is 8

PL/SQL procedure successfully completed.
```

WHILE 循环的一般执行流程如图 6-3 所示。

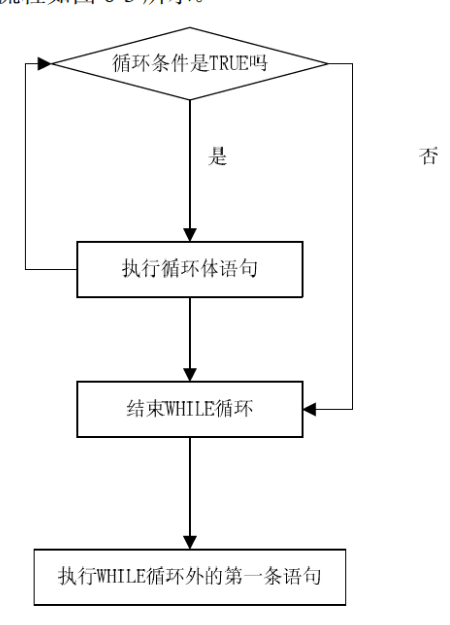


图 6-3　WHILE 循环执行流程

WHILE 循环的关键是：理解一个循环条件的判断，如果满足，则执行循环体语句，当然在循环体中也可以设置条件，从而提前结束循环；如果不满足循环条件，则直接退出循环，执行 WHILE 循环体之外的第 1 条可以执行的语句。

6.3.3　FOR 循环语句

循环控制语句在 PL/SQL 编程中十分常见，使用 FOR 循环可以极大提高数据的操作效率，FOR 循环分为范围 FOR 循环语句和游标 FOR 循环语句。下面我们通过实例介绍这两种 FOR 循环。

1. 范围 FOR 循环语句

范围 FOR 循环语句在外层循环中经常使用 i 作为循环变量，使用关键字 IN 指定循环的范围，因此循环变量 i 就在循环范围内以 1 步进，即 i 每次循环一次加 1。为了演示范围 FOR 循环语句的应用，我们先创建一个测试表，使用 FOR 循环语句向表中插入数据，代码如下。

```
SQL> create table student(id number,name varchar2(20));

Table created.
```

然后使用简单的 LOOP 循环来插入数据，如实例 6-15 所示。

实例 6-15　插入数据。

```
SQL> declare
  2  var_counter number;
  3  begin
  4   for i in 1..1000 loop
  5      insert into student values(i,'name'||to_char(i));
  6    end loop;
  7    commit;
  8  select count(*) into var_counter from student;
  9  dbms_output.put_line('var_counter is :'||var_counter);
 10* end;
SQL> /
var_counter is :1000

PL/SQL procedure successfully completed.
```

粗体部分是使用简单循环插入数据的代码，循环次数为 1000，对于 i 的值从 1~1000 循环向表 student 中插入数据。

前面已经说过，在范围 FOR 循环语句中，外层循环经常使用 i 作为循环变量，而在内层循环中往往使用 j、k 作为循环变量。循环语句的结束由范围控制，一旦遍历完范围空间，则将退出 FOR 循环语句，继续执行 FOR 循环外的第 1 个可执行语句。

范围索引即为包含 IN 之后的起始值和结束值，对其说明如实例 6-16 所示。

实例 6-16　范围索引。

```
SQL>begin
  2   for i in 1..5 loop
  3     dbms_output.put_line(to_char(i));
  4   end loop;
  5* end;
SQL> /
1
2
3
4
5

PL/SQL procedure successfully completed.
```

FOR 的范围索引是 1~5，此时我们依次打印范围索引 i 的值，可以看到输出值为 1~5，从而验证了范围索引是包含边界的。

2．游标 FOR 循环语句

在查询数据库中的表时，使用游标 FOR 循环语句往往很有效，该循环语句使用隐式游标实现，该隐式游标对应 FOR 循环语句的 SELECT 语句部分。使用游标 FOR 循环语句的语法格式如下所示。

```
FOR 游标索引 IN （SELECT 语句） LOOP
     循环执行语句;
END LOOP;
```

下面我们通过实例 6-17 说明如何使用游标 FOR 循环。

实例 6-17　使用游标 FOR 循环。

```
SQL> begin
  for i in (select ename,sal,deptno from emp where sal>2500) loop
    dbms_output.put_line(i.ename||' salary is :'||i.sal||'and in '||i.deptno);
  end loop;
end;
/
JONES salary is :2975  and in 20
BLAKE salary is :2850  and in 30
SCOTT salary is :3000  and in 20
KING salary is :5000  and in 10
FORD salary is :3000  and in 20

PL/SQL procedure successfully completed.
```

在上例中，我们使用游标 FOR 循环来获得 SELECT 语句中返回的一组记录，并使用游标循环来逐一输出 SELECT 语句获得的记录集。这里我们使用游标索引 i 来获取一个记录的某列的值。循环语句会从游标逐一获得记录数据，并打印在控制台中。

6.4　顺序控制语句

顺序控制语句用于完成程序流程的顺序控制，使得程序在某个语句执行完毕后跳转到其他部分，其功能与大多数高级语言中的顺序控制语句类似，这些语句包括 CONTINUE 语句、GOTO 语句。下面分别介绍这两种语句。

6.4.1　CONTINUE 语句

在 Oracle 11g 中，CONTINUE 语句的一个新特性是 CONTINUE 条件。它有两种形式：一种是 CONTINUE；另一种是 CONTINUE WHEN。

1．CONTINUE 语句形式

该语句可以控制程序流程的走向，使循环在满足某个条件时终止循环的迭代。此时需要使用 IF 语句来执行 CONTINUE 条件判断，从而控制何时退出循环。一旦满足 CONTINUE 的条件，则执行循环体中第 1 条可执行的语句。CONTINUE 的语法如下所示。

```
LOOP
执行语句 1.
……
执行语句 n
IF Continue 条件 THEN
      CONTINUE;
END IF;

EXIT WHEN 结束 Continue 条件;
END LOOP;
```

下面我们通过实例 6-18 演示 CONTINUE 语句的执行。

实例 6-18　CONTINUE 语句的执行。

```
SQL>declare
  2    var_num number :=10;
  3  begin
  4  LOOP
  5     var_num := var_num-1;
  6       if var_num>5 then
  7      dbms_output.put_line('var_num is:'||to_char(var_num)||'in continue...');
  8            CONTINUE;
  9       end if;
 10        if var_num=0 then
 11        dbms_output.put_line('end ...');
 12         exit;
 13       end if;
 14            dbms_output.put_line('var_num is : '|| to_char(var_num));
 15  END LOOP;
 16*  end;
```

在上例中，我们设计了一个类似于计数器的程序：首先声明了一个变量 var_num 并赋予初始值 10，然后在循环体中，将该变量减 1，接下来判断该值的大小，如果 var_num 的值大于 5，则执行 CONTINUE 语句，继续执行循环中的语句；因此 CONTINUE 语句会执行 4 次，分别输出 var_num 的值为 9、8、7、6。当 var_num=5 时，因为不满足第一个 IF 条件语句，所以不会执行之后的 CONTINUE 语句。此时结束第 1 个 IF 语句。进入第 2 个 IF 语句：判断 var_num 是否为 0，如果不是，则继续执行循环，直到 var_num=0 为止，因此在退出 CONTINUE 语句之后，var_num 的输出值会是 5、4、3、2、1。最后结束 LOOP 循环。该程序的执行结果如下。

```
var_num is : 9in continue...
var_num is : 8in continue...
var_num is : 7in continue...
var_num is : 6in continue...
var_num is : 5
var_num is : 4
var_num is : 3
var_num is : 2
var_num is : 1
end ...

PL/SQL procedure successfully completed.
```

从输出结果可以知道，我们对该程序功能的分析是正确的，值得注意的是前 4 个语句是因为 CONTINUE 语句而执行的。而结束 CONTINUE 语句使用了 IF 条件判断。

2. CONTINUE WHEN 语句形式

与 CONTINUE 语句相比，CONTINUE WHEN 语句的功能与其大致类似，唯一的差别是执行判断的方式不同，执行语句的时机存在差异。但是二者都是为了控制 CONTINUE 语句正常结束，而不是因无限循环而空耗 CPU 资源。

CONTINUE WHEN 的语法格式如下所示。

```
LOOP
执行语句 1.
......
执行语句 n
```

```
      CONTINUE WHEN Continue 条件;
EXIT WHEN 结束 Continue 条件;
END LOOP;
```

下面我们修改上面的 CONTINUE 实例，如实例 6-19 所示。

实例 6-19　修改上述实例。

```
SQL>declare
  2    var_num number :=10;
  3   begin
  4   LOOP
  5      var_num := var_num-1;
  6        dbms_output.put_line('var_num is:'||to_char(var_num)||'in continue...');
  7          CONTINUE when var_num>5;
  8        if var_num=0 then
  9        dbms_output.put_line('end ...');
 10         exit;
 11       end if;
 12       dbms_output.put_line('var_num is : '|| to_char(var_num)||'out continue');
 13   END LOOP;
 14   end;
 15*
/
```

在 CONTINUE WHEN 语句中判断条件依然是 var_num>5，注意这里是先执行再判断，也就是说在输出 var_num 之后再判断 var_num 的值是否满足 var_num>5 的条件，如果是，则退出 CONTINUE 语句，执行循环 CONTINUE 之后的语句。如果不是，则继续执行 LOOP 循环中 CONTINUE 之前的语句，因为判断 var_num 的值在执行语句之后，所以在 CONTINUE 之中会输出 var_num 的值为 9、8、7、6、5。这里的 var_num=5 会在 CONTINUE WHEN 判断之前输出。在执行判断语句之后 var_num=5，不满足 WHEN 中的 var_num>5 的条件，所以退出 CONTINUE 语句。继续执行下面的 IF 判断语句，因为在整个 LOOP 循环中变量 var_num 始终可见，所以输出如下所示。

```
SQL> /
var_num is : 9in continue...
var_num is : 8in continue...
var_num is : 7in continue...
var_num is : 6in continue...
var_num is : 5in continue...
var_num is : 5out continue
var_num is : 4in continue...
var_num is : 4out continue
var_num is : 3in continue...
var_num is : 3out continue
var_num is : 2in continue...
var_num is : 2out continue
var_num is : 1in continue...
var_num is : 1out continue
var_num is : 0in continue...
end ...

PL/SQL procedure successfully completed.
```

6.4.2 GOTO 语句

GOTO 语句将程序的控制权转移到某个标记处，并且这种转移是无条件的，是一种强制行为，转移到某个标记后的可执行语句或者 PL/SQL 语句块，并继续执行，如实例 6-20 所示，给出了一个 PL/SQL 语句块并执行该过程。

实例 6-20 使用 GOTO 语句。

```
SQL> declare
  var_label  varchar2(40);
  var_num    number;
begin
  for i in 1..100 loop
    if i mod 2 =0 and i>45 then
      var_num:=i;
      var_label:='find the number is : '||to_char(var_num);
    goto end_search;
    end if;
  end loop;
          var_label:='can not find the number';
 <<end_search>>
dbms_output.put_line(var_label);
end;
/
find the number is : 46

PL/SQL procedure successfully completed.
```

整个程序的逻辑是在 1~100 中搜索大于 45 的偶数，当然在实际应用中不会这么简单，这里只是演示 GOTO 语句的用法。在整个搜索过程中，如果找到该数就停止搜索，否则使用 GOTO 语句跳转到<<end_search>>部分，在本例中，我们搜索到了该数值，所以没有使用 GOTO 语句跳转，如果我们修改以下部分，并执行 GOTO 语句：

```
if i mod 2 =0 and i>101 then
```

显然在 LOOP 循环中，该条件无法满足，执行结果如下所示。

```
can not find the number

PL/SQL procedure successfully completed.
```

此时，由于条件不满足，程序执行 GOTO 语句，跳转到<<end_search>>标记，执行该标记后的语句。

显然，利用 GOTO 语句会令程序的执行流程发生变化，这是一种人工控制流程的行为，如果程序段的嵌套复杂，建议不要使用 GOTO 语句，因为使用该语句往往会引起程序的混乱，不容易控制整个流程。

GOTO 语句中的标记只能在语句块之前或者之后使用，而不能在 PL/SQL 语句中间使用，例如实例 6-21 就是错误的。

实例 6-21 标记的错误应用。

```
SQL>declare
```

```
  2   var_b boolean;
  3  begin
  4     for i in 1..100 loop
  5      if var_b then
  6      goto end_search;
  7      end if;
  8      <<end_search>>
  9     end loop;
 10* end;
SQL> /
   end loop;
   *
ERROR at line 9:
ORA-06550: line 9, column 4:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
( begin case declare exit for goto if loop mod null raise
return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue close current delete fetch lock insert open rollback
savepoint set sql execute commit forall merge pipe purge
```

在上例中，标记在 LOOP 循环之内出现，这是不允许的。

6.5　NULL语句

NULL 语句是一种判断语句，其含义是“什么都是，什么都不是”，这样说读者不要奇怪，其实这样说并不矛盾。NULL 意味着无，即使是两个 NULL 也不相等，它们不对应任何具体的东西，有点像我国老子哲学中讲的“非常道”中“道”的意思，当然这只是笔者的比喻，读者不要做严格的学术意义上的思考。

在程序流程控制中 NULL 语句意味着什么也不做，如实例 6-22 所示。

实例 6-22　应用 NULL 语句。

```
SQL> l
  1   declare
  2     var_first_name employees.first_name%type;
  3     var_last_name  employees.last_name%type;
  4     var_employee_id employees.employee_id%type := &employee_id;
  5   begin
  6     select first_name,last_name into
  7     var_first_name,var_last_name
  8     from employees
  9       where employee_id=var_employee_id;
 10     if var_first_name='Stephen' then
 11       dbms_output.put_line('find  data');
 12       dbms_output.put_line('var_first_name : '||var_first_name);
 13       dbms_output.put_line('var_last_name : '||var_last_name);
 14     else
 15       null;
 16     end if;
 17*  end;
```

在上例中，我们通过用户给出的 employee_id 从表 employees 中搜索记录，如果该记录的

var_first_name='Stephen'，则输出其信息，如果不是，则什么也不做，即 NULL，程序流程继续执行。下面我们运行该过程。

```
SQL> /
Enter value for employee_id: 138
old   4:     var_employee_id employees.employee_id%type := &employee_id;
new   4:     var_employee_id employees.employee_id%type := 138;
find  data
var_first_name : Stephen
var_last_name : Stiles

PL/SQL procedure successfully completed.
```

显然，我们找到了匹配条件的记录，即 employee_id 为 138 的员工，其 var_first_name 为 Stephen，所以执行 THEN 后的语句。打印这些信息，下面是没有发现匹配记录的过程。

```
SQL> /
Enter value for employee_id: 200
old   4:     var_employee_id employees.employee_id%type := &employee_id;
new   4:     var_employee_id employees.employee_id%type := 200;

PL/SQL procedure successfully completed.
```

在上述代码中，employee_id 为 200 的记录不包含匹配的数据，所以执行 NULL，即什么也不做便退出程序。

如果我们在 ELSE 后不使用 NULL 语句，而是在控制台打印一行信息，同时我们没有发现匹配数据，此时会发生什么呢？读者或许已经想到了，这显然是一个异常，而我们没有编写异常处理语句，所以会触发 Oracle 内部自定义的异常，可做如下修改。

```
 10     if var_first_name='Stephen' then
 11       dbms_output.put_line('find  data');
 12       dbms_output.put_line('var_first_name : '||var_first_name);
 13       dbms_output.put_line('var_last_name : '||var_last_name);
 14     else
 15       dbms_output.put_line('find no date');
 16     end if;
```

下面我们执行修改过的程序，输入 employee_id 为 2000，该记录不存在，会触发异常。执行结果如下所示。

```
SQL> /
Enter value for employee_id: 2000
old   4:     var_employee_id employees.employee_id%type := &employee_id;
new   4:     var_employee_id employees.employee_id%type := 2000;
 declare
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 6
```

通过上面的实例，可以发现在使用 NULL 的同时也实现了屏蔽异常的作用，即在我们不需要处理异常的场合就可以使用 NULL 不做任何事情。但是通常情况下，还是要提供异常处理功能，这样对于软件的使用者来说就提供了良好的对话功能，即知道发生异常时，大概是什么原因，因为使用者是不知道程序内部情况的，并且绝大多数的使用者都是非专业人员，程序员在编写程序时处

理好异常可以极大地提高软件的友好性。

6.6　本章小结

本章介绍了 PL/SQL 的程序流程控制语句，无论是在面向对象语言中还是面向过程语言中，这类语句都是不可或缺的。因为即使在面向对象语言中依然需要在函数中定义过程控制语句。在面向过程的语言中更是明显，用来控制程序执行的过程。本章介绍了 IF 语句、CASE 语句、循环控制语句以及顺序控制语句，读者只需要明白书中的实例，并仔细琢磨其中的细微差别，就可以在编程时合理使用程序控制语句来控制程序流程。

第 7 章

游 标

游标是 PL/SQL 语言的特殊产物，因为使用 SQL 并不能实现每个应用程序所需的功能，所以此时会使用 PL/SQL 编程实现，一旦使用 PL/SQL，则使用游标就是必然的事情。使用游标要坚持简单化原则，游标用来获取从数据库读取的多行数据，分为静态游标和动态游标，静态游标又分为显式游标和隐式游标，本章我们将重点介绍游标的基础知识。

7.1 显式游标

游标是一个指针，它指向一块 SQL 区域，该区域用于存储处理过来的 SELECT 或其他 DML 操作返回的数据。会话游标在会话结束前保持活跃状态，由 PL/SQL 创建并管理的游标称为隐式游标，由用户创建并管理的游标称为显示游标。我们可以通过游标属性获得会话的相关信息。可通过数据字典视图 v$open_cursor 查看当前的会话游标，这些游标是用户当前打开并解析过的。

在使用 SQL 的 SELECT 语句查询数据时，往往会返回一组记录的集合，为了依次处理记录集合中的每条记录，Oracle 可使用游标来完成，从而遍历每个记录的功能。其实，游标可以看作是指向记录集合的指针，它可以在集合记录中移动以访问每条记录，如图 7-1 所示。

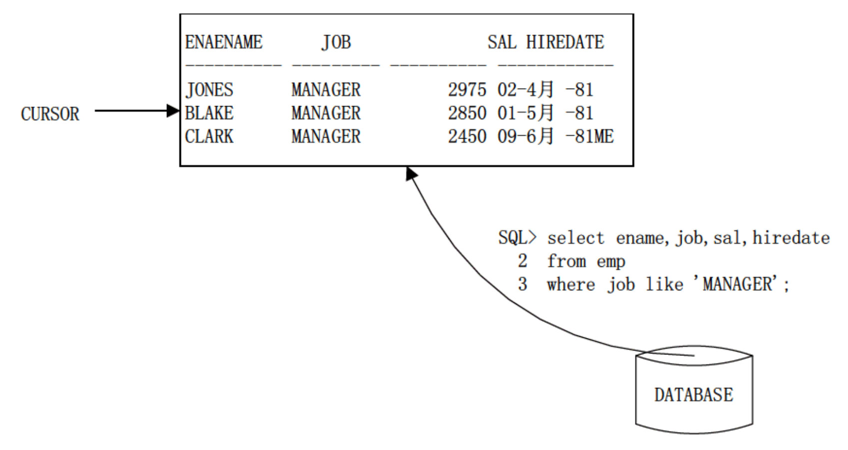


图 7-1 游标的作用示意图

在图 7-1 中，用户选择表 EMP 中的 JOB 为 MANAGER 的记录，而该查询结果有三条记录，通过使用 LOOP 循环语句，再结合使用游标就可以遍历整个查询记录集合。

7.1.1　显示游标的使用详解

1. 创建游标

创建游标的语法如下所示。

```
CURSOR cursor_name IS sql_statements;
```

这里的 CURSOR 声明的就是游标，而随后是用户自定义的游标名字，在关键字 IS 之后是与游标相关联的 SQL 语句。下面给出创建游标的实例 7-1。

实例 7-1　创建游标。

```
CURSOR cselectemp
IS
  select ename,job,sal
   from scott.emp;
```

此时创建了一个游标，游标名为 cselectemp，与该游标相关联的 SQL 语句为“select ename,job,sal from scott.emp;”。使用游标 cselectemp 就可以遍历查询结果中的记录集合。一旦游标创建成功，Oracle 将为其分配内存，并与定义的 SQL 语句关联起来。

2. 打开游标

打开游标的语法如下：

```
OPEN cursor_name [argument [,argument……]];
```

打开游标的内部操作是：Oracle 执行与游标创建时相关联的 SQL 语句，OPEN 是关键字，随后是游标名，此时可以提供参数，这些参数用于传递给为游标创建语句的任何值。打开游标 cselectemp 的内部行为就是执行 SQL 语句。

3. 获取数据

游标可以遍历整个记录集合，依次访问这些记录，其语法如下所示。

```
FETCH cursor_name INTO variable [,variable];
```

执行上述指令时，只能获取记录集合中的一行记录，将这行记录放入随后的变量中，这些变量的数据类型必须与记录中每列的数据类型相同。

在实际中都是使用 LOOP 循环来实现记录集合的遍历，例如我们使用 LOOP…EXIT WHEN 循环实现，如实例 7-2 所示。

实例 7-2　利用游标获取数据。

```
LOOP
 FETCH cselectemp
 INTO employeename,employeejob,employeesal;
 EXIT WHEN cselectemp%notfound;
DBMS_OUTPUT.put_line('employeename is '
                      ||employeename
                      ||'employeeejob is '
                      ||employeejob
                      ||'employeesal is'
```

```
                        ||employeesal );
END LOOP;
```

循环结束的条件就是游标的属性值%NOTFOUND 为真，即游标已经遍历完所有记录，每次循环将读取的记录信息从标准输入输出装置中输出显示。

4. 关闭游标

关闭游标的语法如下：

```
CLOSE cursor_name
```

一旦关闭游标，则 Oracle 将释放为其分配的内存，游标不能重复打开，在打开前必须先将其关闭。

5. 游标属性

游标一旦打开，就将处于某种状态中，为了了解游标的状态，以及使用这些状态实现某些逻辑操作，我们需要知道游标的属性，代码如下所示。

- %ISOPEN：判断游标是否打开。
- %FOUND：游标发现数据。
- %NOTFOUND：游标没有发现数据。
- %ROWCOUNT：游标可以遍历的记录的数量。

访问游标属性的语法如下。

```
cursor_name.attribute_name
```

7.1.2 显式游标的使用实例

前面已经讲解了与游标相关的所有语法，下面我们通过一个具体的实例来说明创建和使用游标的全过程，创建一个过程，在该过程中使用游标来遍历查询的数据记录集合。下面这个实例采用步进的方式，从而不断完善整个过程的程序代码。

1. 声明变量

首先创建一个过程，该过程的声明如下。

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
```

过程名为 cursortest，该过程声明了三个变量，即 employeename、employeejob 和 employeesal，分别与用户 scott 的 EMP 表的三个列（即 ENAME、JOB 和 SAL）对应。

2. 声明游标

接着，我们声明游标变量，即创建游标，代码如下所示。

```
create or replace procedure cursortest
is
```

```
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
cursor cselectemp
is
  select ename,job,sal
   from scott.emp;
```

这里的游标声明在过程的声明后进行，该游标的名字为 cselectemp，与之关联的 SQL 语句为“select ename,job,sal from scott.emp;”，此时 Oracle 为游标 cselectemp 分配内存。

3. 打开游标

在声明游标后，需要执行该游标以读取数据，所以接着打开游标，代码如下所示。

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
cursor cselectemp
is
  select ename,job,sal
   from scott.emp;
begin
open cselectemp;
```

打开游标在过程的执行区进行，一旦打开游标，Oracle 将立即执行与之相关联的 SQL 语句，而后可以使用游标读取数据。

4. 获取数据

因为游标访问的数据是一个记录集合，所以往往需要一个循环语句来完成整个记录的访问。此时，我们使用一个 LOOP…EXIT WHEN 来循环访问数据记录的集合。

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
cursor cselectemp
is
  select ename,job,sal
   from scott.emp;
begin
open cselectemp;
loop
 fetch cselectemp
  into employeename,employeejob,employeesal;
 exit when cselectemp%notfound;
 dbms_output.put_line('employeename is '
                    ||employeename
                    ||'employeeejob is '
                    ||employeejob
                    ||'employeesal is'
                    ||employeesal );
end loop;
```

```
end;
```

在 LOOP 循环中，如果游标在移动过程中发现记录存在，就继续执行循环，并把数据打印到标准输出装置。一旦游标移动后不能发现记录存在就结束循环，注意此时我们使用了游标的属性%NOTFOUND。但是读者或许已经注意到我们没有释放游标，游标是占用内存的，所以使用之后必须关闭游标以释放内存资源。

5. 释放资源

最后，我们关闭游标，以完成整个包含游标的过程的创建，如实例 7-3 所示。

实例 7-3　在存储过程中创建游标的完整示例。

```
create or replace procedure cursortest
is
  employeename varchar2(20);
  employeejob  varchar2(9);
  employeesal  number(7,2);
cursor cselectemp
is
  select ename,job,sal
   from scott.emp;
begin
open cselectemp;
loop
 fetch cselectemp
  into employeename,employeejob,employeesal;
 exit when cselectemp%notfound;
 dbms_output.put_line('employeename is '
                     ||employeename
                     ||'employeeejob is '
                     ||employeejob
                     ||'employeesal is'
                     ||employeesal );
end loop;
close cselectemp;
end;
```

我们将该代码在记事本中编辑，并保存为后缀为.SQL 的脚本文件，然后在 SQL*Plus 中的 scott 用户下编译该脚本、创建过程 cselectemp，下面我们执行该过程并查询结果。

6. 运行包含游标的存储过程

在 scott 模式下创建完过程 cursortest 后即可执行该过程，该过程在执行中使用游标遍历访问 EMP 表的数据记录集合，然后输入这些行记录，直到游标的%NOTFOUND 值为真时，则结束对记录集合的遍历。执行存储过程如实例 7-4 所示。

实例 7-4　执行存储过程 cursortest。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> set serveroutput on;
SQL> execute cursortest
employeename is tomemployeeejob is SALESMANemployeesal is2000
employeename is SMITHemployeeejob is CLERKemployeesal is800
employeename is ALLENemployeeejob is SALESMANemployeesal is1600
```

```
employeename is WARDemployeeejob is SALESMANemployeesal is1250
employeename is JONESemployeeejob is MANAGERemployeesal is2975
employeename is MARTINemployeeejob is SALESMANemployeesal is1250
employeename is BLAKEemployeeejob is MANAGERemployeesal is2850
employeename is CLARKemployeeejob is MANAGERemployeesal is2450
employeename is SCOTTemployeeejob is ANALYSTemployeesal is3000
employeename is KINGemployeeejob is PRESIDENTemployeesal is5000
employeename is TURNERemployeeejob is SALESMANemployeesal is1500
employeename is ADAMSemployeeejob is CLERKemployeesal is1100
employeename is JAMESemployeeejob is CLERKemployeesal is950
employeename is FORDemployeeejob is ANALYSTemployeesal is3000
employeename is MILLERemployeeejob is CLERKemployeesal is1300

PL/SQL 过程已成功完成。
```

7.2　隐式游标

隐式游标是没有声明的游标，类似于在 Java 语言中的无名内隐类的概念。没有显式地创建该游标，只是在 PL/SQL 代码块中执行 SQL 语句，这些 SQL 语句就是隐式游标，隐式游标也有属性，如 SQL%ROWCOUNT，即该 SQL 语句返回的记录数量，下面给出一个实例来说明如何使用隐式游标，以及隐式游标的属性。修改实例 7-3，使用隐式游标记录表 EMP 中不同的工作岗位的数量，并打印到标准输出装置，如实例 7-5 所示。

实例 7-5　使用隐式游标的示例。

```
create or replace procedure hidencursortest
is
jobnumber NUMBER;
cursor cselectemp
is
     select ename,job,sal
     from scott.emp;
begin
select count(distinct(job))
into jobnumber
from emp;
dbms_output.put_line('there are '
                   ||jobnumber
                   ||'diferent jobs');
dbms_output.put_line('hiden cursor rowcount is '
                   ||SQL%ROWCOUNT);
for emp_record in cselectemp
loop
 dbms_output.put_line('employeename is '
                   ||emp_record.ename
                   ||'employeeejob is '
                   ||emp_record.job
                   ||'employeesal is'
                   ||emp_record.sal );
end loop;
end;
```

在上例中，执行区开始（begin）后使用了 SQL 语句，用于查询表 EMP 中不同工作岗位的数

量，并保存到过程声明的变量 jobnumber 中，然后打印到输出装置。此处的 SQL 语句就是一个隐式游标，接着我们输出了隐式游标执行的 SQL 语句返回的记录数，从而说明了隐式游标的存在。在记事本中编辑该文件，并保存在 F 盘的根目录下，保存的文件名为 hidencursortest.sql。编译并执行该过程，代码如下所示。

```
SQL>  @f:\hidencursortest.sql
 27  /

过程已创建。
```

下面执行过程 hidencursortest，并观察输出结果，如实例 7-6 所示。

实例 7-6　执行过程 hidencursortest。

```
SQL> execute hidencursortest
there are 5diferent jobs
hiden cursor rowcount is 1
employeename is tomemployeeejob is SALESMANemployeesal is2000
employeename is SMITHemployeeejob is CLERKemployeesal is800
employeename is ALLENemployeeejob is SALESMANemployeesal is1600
employeename is WARDemployeeejob is SALESMANemployeesal is1250
employeename is JONESemployeeejob is MANAGERemployeesal is2975
employeename is MARTINemployeeejob is SALESMANemployeesal is1250
employeename is BLAKEemployeeejob is MANAGERemployeesal is2850
employeename is CLARKemployeeejob is MANAGERemployeesal is2450
employeename is SCOTTemployeeejob is ANALYSTemployeesal is3000
employeename is KINGemployeeejob is PRESIDENTemployeesal is5000
employeename is TURNERemployeeejob is SALESMANemployeesal is1500
employeename is ADAMSemployeeejob is CLERKemployeesal is1100
employeename is JAMESemployeeejob is CLERKemployeesal is950
employeename is FORDemployeeejob is ANALYSTemployeesal is3000
employeename is MILLERemployeeejob is CLERKemployeesal is1300

PL/SQL 过程已成功完成。
```

在该过程中，我们看到隐式游标统计了表 EMP 中不同岗位的数量，并且输出了隐式游标中的记录行数量，该值为 1，符合隐式游标执行的 SQL 语句的返回结果，即函数 COUNT(DISTINCT(JOB)) 返回一行记录。

在用户每次运行 SELECT 以及 DML 操作时，PL/SQL 会自动创建一个隐式游标。隐式游标无法控制，一旦游标涉及的 SQL 语句执行结束，隐式游标也将自动关闭。但是可以通过隐式游标的属性获得隐式游标的信息。

隐式游标也被称为 SQL 游标，所以隐式游标的属性的语法格式为 SQL attribute，如 SQL%ISOPEN，用于判断游标是否打开。

下面是隐式游标的属性。

1. SQL%ISOPEN

SQL%ISOPEN 用于判断游标是否打开。SQL%ISOPEN 总是返回 FALSE，因为隐式游标在运行相关语句后总是将其关闭。

2. SQL%FOUND

SQL%FOUND 用于发现游标中的数据，可根据返回值来判断数据的返回情况，返回值如下。

- NULL：没有 SELECT 或者 DML 语句运行时。
- TRUE：SELECT 语句返回一行或者多行数据，以及 DML 语句影响了一行或者多行数据。
- FALSE：其他情况。

下面我们给出实例 7-7，可通过实例理解该属性的含义，即在 HR 用户模式下创建一个测试表。

实例 7-7　创建测试表。

```
DROP TABLE departments_test;
CREATE TABLE deparments_test AS
SELECT * FROM departments;
```

此时，我们创建了一个测试表 departments_test，然后测试在 PL/SQL 语句块中隐式游标的属性值，如实例 7-8 所示。

实例 7-8　创建过程 ptestfound。

```
drop procedure ptestfound;
CREATE OR REPLACE PROCEDURE ptestfound (
   dept_no NUMBER
  ) AUTHID DEFINER AS
    BEGIN
      DELETE FROM departments_test
        WHERE department_id = dept_no;
     IF SQL%FOUND THEN
       DBMS_OUTPUT.PUT_LINE (
           'Delete succeeded for department numb' || dept_no);
     ELSE
       DBMS_OUTPUT.PUT_LINE ('No department number' || dept_no);
     END IF;
   END;
 /
```

以上代码创建了一个过程 ptestfound，在 PL/SQL 语句块中的执行部分使用了一个 DML 操作，然后通过隐式游标属性 SQL%FOUND 来测试是否发现该语句影响的数据，如果有，则打印信息。该过程包含一个参数 dept_no，用于删除表 departments_test 中对应的行。

从上述粗体字部分也可以看出，一旦在 PL/SQL 语句块中执行了 DML 操作，PL/SQL 将自动创建并维护该游标，显然这个游标是不可见的，我们没有明确定义，可直接使用 SQL%FOUND 来获取游标属性，下面执行该过程，如实例 7-9 所示。

实例 7-9　执行过程 ptestfound。

```
SQL> begin
 ptestfound(260);
 ptestfound(280);
end;
/
Delete succeeded for department numb260
No department number280
```

```
PL/SQL procedure successfully completed.
```

下面我们分析第 1 个调用：输入给过程 ptestfound 的参数是 260，也就是说需要删除表 departments_test 中 employee_id 为 260 的记录，因为该记录存在，所以 DELETE 语句执行成功，DELETE 语句影响了一行数据，此时的隐式游标属性 SQL%FOUND 的值为 TRUE，则 IF-THEN 判断语句成功执行，并将继续执行 THEN 之后的语句，所以输出显示被删除记录的 department_id 的值。

3. SQL%NOTFOUND

该属性和 SQL%FOUND 正好相反，其取值如下所示。

- NULL：没有 SELECT 或者 DML 语句执行。
- FLASE：SELECT 语句返回了数据行，或者 DML 语句影响了数据行。
- TRUE：其他。

SQL%NOTFOUND 属性对于 PL/SQL 的 SELECT INTO 语句没有用处，PL/SQL 认为其是一个非法操作。因为如果发生 SELECT INTO 没有数据返回的情况，将会触发一个预定义异常 NO_DATA_FOUND。

下面通过实例 7-10 了解该属性的含义。

实例 7-10　验证 SQL%NOTFOUND 属性。

```
SQL> declare
sum number;
loc_id number := &location_id;
begin
  select count(*) into sum from departments where location_id=loc_id;
    if SQL%NOFOUND THEN
         DBMS_OUTPUT.PUT_LINE ('NO DATA FOUND!');
    else
         DBMS_OUTPUT.PUT_LINE ('sum is : ');
    end if;
end;
.
/
```

在上述匿名 PL/SQL 语句块中，计算表 departments 中具有某一相同 location_id 的员工数量，如果 SELECT INTO 没有值，可使用隐式游标的 SQL%NOTFOUND 属性尝试处理这个异常，执行情况如下所示。

```
Enter value for location_id: 12
old   3: loc_id number := &location_id;
new   3: loc_id number := 12;
    if SQL%NOFOUND THEN
          *
ERROR at line 6:
ORA-06550: line 6, column 12:
PLS-00207: identifier 'NOFOUND', applied to implicit cursor SQL, is not a legal
cursor attribute
ORA-06550: line 6, column 5:
PL/SQL: Statement ignored
```

显然，从输出可以知道使用 SQL%NOTFOUND 本身就是非法操作。下面我们新建一个匿名过程，看看 PL/SQL 的 SELECT INTO 是如何处理 NO_DATA_FOUND 错误的，如实例 7-11 所示。

实例 7-11　通过异常处理 NO_DATA_FOUND 错误。

```
SQL> declare
name employees.first_name%type;
loc_id number := &location_id;
begin
  select first_name into name from employees where department_id=loc_id;
   exception
     when no_data_found then
        dbms_output.put_line('no data found');
end;
.
/
```

在上例的异常处理中，我们没有使用 SQL%NOTFOUND 属性来判断是否存在返回数据，因为这样的操作是非法的，所以我们使用预定义的异常 NO_DATA_FOUND 来处理没有返回数据的操作，执行结果如下所示。

```
Enter value for location_id: 12
old   3: loc_id number := &location_id;
new   3: loc_id number := 12;
no data found

PL/SQL procedure successfully completed.
```

此时，我们输入了 department_id 的值为 12，而表 employees 中根本没有该值的记录，所以触发了 NO_DATA_FOUND 异常，程序流程跳转到了 EXCEPTION 部分，即可执行 THEN 之后的语句。

4. SQL%ROWCOUNT

SQL%ROWCOUNT 用于发现游标中涉及的返回结果的行数量。

为了演示该属性的作用，我们先创建一个测试表 employees_test1，如实例 7-12 所示。

实例 7-12　创建测试表 employees_test1。

```
SQL> create table employees_test1 as  select * from employees;

Table created.
```

接下来，我们创建一个匿名过程，删除表 employees_test1 中指定 manager_id 的所有记录，如实例 7-13 所示。

实例 7-13　创建匿名过程。

```
SQL> declare
 deleted_sum  number;
 mgr_id   number := &mgr_id;
begin
 delete from employees_test1 where manager_id = mgr_id;
 dbms_output.put_line('employees deleted is :'||to_char(SQL%ROWCOUNT));
END;
.
/
```

在上例中，我们使用替代变量&mgr_id 来输入用户需要删除的记录。下面是执行该匿名过程的结果。

```
Enter value for mgr_id: 122
old   3:  mgr_id   number := &mgr_id;
new   3:  mgr_id   number := 122;
employees deleted is :8

PL/SQL procedure successfully completed.
```

从输出结果可以知道 DELETE 语句删除了 8 行数据。如果 DELETE 语句没有删除数据，则属性 SQL%ROWCOUNT 的值为 0。

7.3 FOR游标

使用 FOR 游标可以简化实例 7-3 中的代码，使用 FOR 游标不需要声明变量，也不需要显式地打开游标和关闭游标，这些都是自动执行的。使用 FOR 游标的语法如下所示。

```
FOR record IN cursor_name
LOOP
 Logical statements……
END LOOP;
```

在 FOR 游标的语法中，record 为一条记录的集合，它自动定义为一个%ROWTYPE 类型的变量，%ROWTYPE 变量包含对应于记录中的多列变量，通过这个变量可以依次访问该记录中的每个列值。LOOP 循环将自动遍历游标所涉及的记录集合，循环开始时打开游标，而当循环结束时，游标自动关闭。下面我们给出改写实例 7-3 中创建过程的示例，如实例 7-14 所示。

实例 7-14　使用 FOR 游标的示例。

```
create or replace procedure forcursortest
is
cursor cselectemp
is
       select ename,job,sal
       from scott.emp;
begin
for emp_record in cselectemp
loop
 dbms_output.put_line('employeename is '
                        ||emp_record.ename
                        ||'employeeejob is '
                        ||emp_record.job
                        ||'employeesal is'
                        ||emp_record.sal );
end loop;
end;
```

同样，将上述文件在记事本中编辑，保存在 F 盘的根目录下，命名为 forcursortest.sql 文件，然后编译该文件，创建过程 forcursortest，代码如下所示。

```
SQL> conn scott/oracle@orcl
已连接。
```

```
SQL> @f:\forcursortest.sql;
 2  /

过程已创建。
```

可通过数据字典 USER_PROCEDURES 验证是否创建了过程 forcursortest，如实例 7-15 所示。

实例 7-15　检验过程 forcursortest 是否已创建成功。

```
SQL> select object_name
  2  from user_procedures
  3  where object_name like '%TEST';

OBJECT_NAME
------------------------------
GOTOTEST
PROTEST
CURSORTEST
FORCURSORTEST
```

由结果可见，过程 forcursortest 创建成功，下面我们执行该过程，执行结果如下。

```
SQL> set serveroutput on
SQL> execute forcursortest
employeename is tomemployeeejob is SALESMANemployeesal is2000
employeename is SMITHemployeeejob is CLERKemployeesal is800
employeename is ALLENemployeeejob is SALESMANemployeesal is1600
employeename is WARDemployeeejob is SALESMANemployeesal is1250
employeename is JONESemployeeejob is MANAGERemployeesal is2975
employeename is MARTINemployeeejob is SALESMANemployeesal is1250
employeename is BLAKEemployeeejob is MANAGERemployeesal is2850
employeename is CLARKemployeeejob is MANAGERemployeesal is2450
employeename is SCOTTemployeeejob is ANALYSTemployeesal is3000
employeename is KINGemployeeejob is PRESIDENTemployeesal is5000
employeename is TURNERemployeeejob is SALESMANemployeesal is1500
employeename is ADAMSemployeeejob is CLERKemployeesal is1100
employeename is JAMESemployeeejob is CLERKemployeesal is950
employeename is FORDemployeeejob is ANALYSTemployeesal is3000
employeename is MILLERemployeeejob is CLERKemployeesal is1300

PL/SQL 过程已成功完成。
```

从过程 forcursortest 的输出结果可以看出，使用 FOR 游标极大地简化了编码，从而减少了代码量。

7.4　游标变量

在 PL/SQL 程序块内，游标与静态 SQL 绑定在一起，此时的游标变量只是指向游标的结果集。而使用游标变量将使得在 PL/SQL 语句块中的游标变量可以与任意的游标相结合，这样就提高了 PL/SQL 语句块中游标的灵活性。

下面先创建一个函数，用于返回游标，如实例 7-16 所示。

实例 7-16　创建函数 cursoremp。

```
SQL> create or replace function cursoremp
  return sys_refcursor
```

```
    is
      empcursor sys_refcursor;
   begin
     open empcursor for select * from employees where salary>13000;
       return empcursor;
   end;
   /

Function created.
```

此时我们创建了一个函数 cursoremp，该函数返回一个游标，该游标指向 SELECT 语句返回的数据区，用来获取 SQL 语句返回的数据。

当然，我们也可以创建其他功能的函数，用于返回不同的游标，这样的游标可以被其他游标变量调用，即给游标变量赋值，如实例 7-17 所示。

实例 7-17　创建过程 print_emp。

```
SQL> create or replace procedure print_emp is
    var_cursor  sys_refcursor;
    var_row     employees%rowtype;
begin
  var_cursor := cursoremp;
   loop
      fetch var_cursor into var_row;
       if var_cursor%notfound then
           exit;
       end if;
      dbms_output.put_line(var_row.first_name||' : '||var_row.salary);
   end loop;
 close var_cursor;
end;
  /

Procedure created.
```

在上例中，我们先定义一个变量 var_cursor，在过程的执行部分使用语句“var_cursor := cursoremp;”，函数 cursoremp 返回一个游标，使用该函数返回的游标来为游标变量 var_cursor 赋值。

我们使用 IF-THEN 语句来控制 LOOP 循环结束，这里使用了游标属性%NOTFOUND，并打印 SELECT 语句返回的部分信息。下面我们执行该过程。

```
SQL> exec print_emp;
Steven : 24000
Neena : 17000
Lex : 17000
John : 14000
Karen : 13500

PL/SQL procedure successfully completed.
```

print_emp 过程存储在数据库服务器上，客户端 SQL*Plus 中的客户端程序可以获取服务器上的过程返回的游标，即获取来自游标变量的值。通过对游标的 PRINT 指令获得了游标所指向的数据。这样在客户端和数据库服务器的存储过程中通过游标变量将其联系起来。下面我们在 SQL*Plus 客户端定义一个变量：

```
SQL> variable var_sqlcur refcursor;
```

此时，我们在客户端定义了一个游标变量 var_sqlcur。下面我们再通过实例 7-18 调用数据库服务器端的存储过程和函数。这样在客户端和服务器之间就建立了连接。

实例 7-18　调用服务器端的存储过程。

```
SQL> declare
  var_row employees%rowtype;
begin
  :var_sqlcur := cursoremp;
  loop
    fetch :var_sqlcur into var_row;
    exit  when  :var_sqlcur%notfound;
      dbms_output.put_line(var_row.first_name||' : '||var_row.salary);
  end loop;
 close :var_sqlcur;
end;
/
Steven : 24000
Neena : 17000
Lex : 17000
John : 14000
Karen : 13500

PL/SQL procedure successfully completed.
```

在实例 7-19 中，我们在客户端调用服务器端的函数 cursoremp，然后将获得的游标值赋予客户端的游标变量 var_sqlcur。

实例 7-19　调用服务器端的函数 cursoremp。

```
SQL> exec :var_sqlcur := cursoremp;

PL/SQL procedure successfully completed.
```

下面我们就可以使用 PRINT 指令打印该游标所获取的数据行，如实例 7-20 所示。

实例 7-20　使用 PRINT 指令打印游标获取的数据。

```
SQL> print var_Sqlcur;

EMPLOYEE_ID FIRST_NAME           LAST_NAME
----------- -------------------- -------------------------
EMAIL                     PHONE_NUMBER         HIRE_DATE JOB_ID         SALARY
------------------------- -------------------- --------- ---------- ----------
COMMISSION_PCT MANAGER_ID DEPARTMENT_ID
-------------- ---------- -------------
       100 Steven               King
SKING                     515.123.4567         17-JUN-03 AD_PRES         24000
                                  90
```



这里省略剩余的输出。

7.5 游标表达式

在 SQL 查询语句中，可以使用声明的变量指向从 SQL 语句获取的数据，该变量实际上就是一个数据指针，指向 SQL 语句返回数据的存储地址。在 SELECT 语句中，我们也可以使用游标表达式来实现变量的作用，如实例 7-21 所示。

实例 7-21　游标表达式。

```
SQL> select deptno,dname,loc,
  2      cursor (select empno,ename,sal
  3                 from emp
  4                 where deptno=d.deptno)
  5  from dept d;
```

在上例中，随着从 EMP 表获取的数据依次打开游标，将获得与谓词匹配的数据，下面是该 SELECT 语句的返回结果，如实例 7-22 所示。

实例 7-22　游标表达式的输出结果。

```
    DEPTNO DNAME          LOC           CURSOR(SELECTEMPNO,E
---------- -------------- ------------- --------------------
        10 ACCOUNTING     NEW YORK      CURSOR STATEMENT : 4

CURSOR STATEMENT : 4

     EMPNO ENAME             SAL
---------- ---------- ----------
      7782 CLARK            2450
      7839 KING             5000
      7934 MILLER           1300

        20 RESEARCH       DALLAS        CURSOR STATEMENT : 4

CURSOR STATEMENT : 4

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
      7566 JONES            2975
      7788 SCOTT            3000
      7876 ADAMS            1100
      7902 FORD             3000

        30 SALES          CHICAGO       CURSOR STATEMENT : 4

CURSOR STATEMENT : 4

     EMPNO ENAME             SAL
---------- ---------- ----------
      7499 ALLEN            1600
      7521 WARD             1250
      7654 MARTIN           1250
      7698 BLAKE            2850
      7844 TURNER           1500
```

```
      7900 JAMES            950

已选择 6 行。

        40 OPERATIONS     BOSTON        CURSOR STATEMENT : 4

CURSOR STATEMENT : 4

未选定行
```

每次从 DEPT 表读取数据时，都会依次打开利用游标获得的与谓词相匹配的数据，如从 DEPT 表中读取 DEPTNO 为 10 的数据时，随着打开游标将从 EMP 表中获得 DEPTNO 为 20 的所有记录。

游标表达式也可以嵌套使用，即在声明的游标的 SQL 语句中使用游标，如实例 7-23 所示。

实例 7-23　使用嵌套游标。

```
SQL> declare
  2    type var_cur_type is ref cursor;
  3    var_cur        var_cur_type;
  4    dept_name  dept.dname%type;
  5    emp_name  emp.ename%type;
  6
  7  cursor  cur is
  8
  9        select dname,
 10          cursor (select e.ename
 11                     from emp e
 12                     where e.deptno=d.deptno) emp
 13         from dept d
 14      where dname like 'R%' ;
 15  begin
 16    open cur;
 17     loop
 18     fetch cur into dept_name,var_cur;
 19     exit when cur%notfound;
 20     dbms_output.put_line('dept_name: '|| dept_name);
 21     loop
 22       fetch  var_cur into emp_name;
 23       exit  when  var_cur%notfound;
 24       dbms_output.put_line('emp_name: '||emp_name);
 25      end loop;
 26     end loop;
 27    close cur;
 28  end;
 29  /
```

在上例中，我们定义了显式游标 cur，在该游标的定义中使用了嵌套游标表达式（主游标的查询用于获得满足谓词的 DNAME 属性值）和对应的游标表达式（游标表达式中的查询用于获得 ENAME），条件是：和主游标查询中的谓词匹配的 DEPTNO 相等的记录。下面是上例中的嵌套游标实例的查询结果。

```
dept_name: RESEARCH
emp_name: SMITH
emp_name: JONES
```

```
emp_name: SCOTT
emp_name: ADAMS
emp_name: FORD

PL/SQL 过程已成功完成。
```

其中，获得的 dept_name 为 RESEARCH，而在表 DEPT 中对应的部门号为 20，所以在游标表达式对应的查询中获得部门号为 20 的所有 DNAME 属性值。

下面我们通过分析表 EMP 和 DEPT 来验证我们的分析：首先在 DEPT 表中查看 DNAME 为 RESEARCH 的部门号，将这个查询结果作为主查询的谓词的一部分，主查询从 EMP 表中获得对应的 ENAME，如实例 7-24 所示。

实例 7-24　使用嵌套查询实现嵌套游标功能。

```
SQL> select ename from emp where
  2  deptno in (select deptno
  3            from dept
  4            where dname='RESEARCH');

ENAME
----------
SMITH
JONES
SCOTT
ADAMS
FORD
```

显然，游标嵌套表达式的实例和通过上例中的嵌套查询的方式获得的数据一致，也与我们分析游标嵌套表达式的执行结果一致。

7.6　动态游标

动态游标，即 REF 游标，它是一个记录的集合，Oracle 允许在不同的程序单元之间传递游标的引用。

可以在存储过程中定义 REF 游标，这样就可以在需要的时候使用该游标，从而不必在需要游标时都显式地定义。若要使用 REF 游标，首先需要声明它为一个 TYPE，声明方式如下所示。

```
TYPE ref 游标名 IS REF CURSOR [返回类型]
```

如 TYPE ty_emprefcur IS REF CURSOR，说明 ty_emprefcur 是 REF 游标。在使用时需要创建该类型的一个实例，代码如下所示。

```
empcursor  ty_emprefcur;
```

此时 empcursor 就是 REF 游标 ty_emprefcur 的一个实例。当然还可以创建其他实例。下面给出一个完整的实例 7-25。

实例 7-25　使用 REF 游标的示例。

```
create or replace procedure refcursortest
```

```
is
    type ty_emprefcur is ref cursor; --声明 REF 游标为一个 TYPE
    empcursor       ty_emprefcur; --创建该类型的一个实例
    rec_emp         emp%rowtype;
    rec_sal         emp.sal%type;
    rec_job         emp.job%type;
begin
  open empcursor for --使用 REF 游标
       select *
         from emp;

  fetch empcursor
        into rec_emp;
  close empcursor;
dbms_output.put_line('employeename is '||rec_emp.ename);

open empcursor for --使用 REF 游标
      select sal
          from emp;
fetch empcursor
   into rec_sal;
dbms_output.put_line('employee sal is '||rec_sal);
close empcursor;

open empcursor for --使用 REF 游标
      select job
         from emp;
 fetch empcursor
   into rec_job;
dbms_output.put_line('employee job is '||rec_job);
end;
/
```

我们在 Windows 的记事本中编辑该文件，并保存在 F 盘的根目录下，文件名为 refcursortest.sql，然后创建一个存储过程，如实例 7-26 所示。

实例 7-26　创建包含 REF 游标的存储过程。

```
SQL> @f:\refcursortest.sql;

过程已创建。
```

当存储过程创建成功之后，我们执行该过程，查看输出结果。

```
SQL> execute refcursortest;
employeename is SMITH
employee sal is 800
employee job is CLERK

PL/SQL 过程已成功完成。
```

在过程 refcursortest 中，有 3 次使用 REF 游标，其实，在打开 REF 游标时就给出了与之关联的 SQL 语句，然后将执行游标时返回的记录集合的第一个记录放入一个变量，然后输入该值。

如果希望 REF 游标返回的数据类型更加严格，可以使用%ROWTYPE，代码如下所示。

```
TYPE empcursor IS REF CURSOR
```

```
    RETURN emp%rowtype
```

7.7　本章小结

本章详细介绍了游标的使用，游标是 PL/SQL 编程中操作数据集的对象。使用游标可以非常灵活地实现对从数据库获取的数据集合的操作，本章介绍了如何创建、打开、读取数据以及关闭游标。游标作为一个可操作的对象，提供了游标属性，使用该属性可以轻松控制程序流程，获取游标状态。随后介绍了隐式游标和显式游标的用法，并且都是通过实例进行演示。最后引入了游标变量和游标表达式，使得读者可以更灵活地使用游标来实现自己的程序逻辑。

第 8 章

触发器

触发器类似于 Oracle 的 PL/SQL 存储过程，但是它不能被显式调用，而是由数据库服务器维护，在特定事件发生时由 Oracle 数据库调用，触发器利用 PL/SQL 语言编写并保存在数据库服务器上。Oracle 提供了在 DML 操作和 DDL 操作之前和之后的触发器，还提供了当数据库关闭与启动或者用户登录与退出的触发器，这些触发器极大地丰富了 DBA 执行、审计、安全或完整性关联的管理任务。

8.1 触发器的创建

本节我们将给出创建触发器的实例，通过实例读者可以直观感受创建触发器的语法，以及如何创建 DML 触发器。

8.1.1 创建标准触发器

创建触发器的语法为：

```
CREATE TRIGGER trigger_name
BEFORE/AFTER DELETE [UPDATE,INSERT,SHUTDOWN……]
ON object_name
Trigger_SQL_PL/SQL_body
```

其中，CREATE TRIGGER 表示要创建一个触发器，随后是触发器名称、触发器的触发时机（即 BEFORE 或 AFTER）、触发事件（如 DML 的操作 DELETE 或数据库关闭操作 SHUTDOWN）、ON 关键字（使用 ON 关键字说明触发器操作的对象，该对象可以是表或者数据库 DATABASE），最后是触发器的主体代码逻辑。

下面我们创建一个触发器，该触发器用于实现对 SCOTT 模式下的表 EMP 的操作记录，即 DML 操作的记录，当发生任何 DML 操作时，都在标准输出装置中打印一条信息。当然这里可以是更复杂的记录，如记录用户删除的数据、记录更新前的原始值、插入数据之前的原始值、记录用户 DML 操作的次数等，总之读者可以根据自己的需求进行设置，如实例 8-1 所示。

实例 8-1　创建一个标准的简单触发器。

```
create or replace trigger delete_emp_trigger
 before delete
      on emp
```

```
 for each row
    begin
      dbms_output.put_line('deleting......');
end delete_emp_trigger;
/
```

将上述代码在记事本中编译为一个后缀为.SQL 的脚本文件，而在 scott 模式下编译并创建触发器 delete_emp_trigger。笔者将该文件保存在 F 盘的根目录下，编译过程如下所示。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> @f:\delete_emp_trigger.sql;

触发器已创建
```

为了验证创建触发器的结果，可以使用数据字典 USER_OBJECTS 进行查询，如实例 8-2 所示。

实例 8-2　使用数据字典 USER_OBJECTS 查询创建的触发器信息。

```
SQL> col object_name for a20
SQL> select object_name,object_type,created,status
  2  from user_objects
  3* where object_type ='TRIGGER'

OBJECT_NAME          OBJECT_TYPE           CREATED     STATUS
------------------ -------------- ----------------------- ----------------
SCOTT_EMP_DML        TRIGGER              19-8 月 -09     VALID
DELETE_EMP_TRIGGER    TRIGGER              19-8 月 -09     VALID
```

由此可见，触发器 DELETE_EMP_TRIGGER 创建成功，且该触发器为有效的触发器，因为该行的 STATUS 列值为 VALID。此时，触发器如同存储过程和函数一样，存储在表中，并与具体的表相关联，在第一次被调用时即时编译。

下面我们将演示触发器的行为，以及 Oracle 数据库何时使用触发器。再强调一下，我们创建的触发器 DELETE_EMP_TRIGGER 的作用是在表 EMP 中删除行记录时，在删除的每一行记录前都触发一个事件，该事件将向标准输出打印一行信息，其实，在现实的生产数据库中，触发器行为逻辑可能会很复杂，这里我们只是给出一个演示，希望读者通过这个简单的触发器行为，理解如何创建触发器和触发器的行为，如实例 8-3 所示。

实例 8-3　通过删除表 EMP 中的记录来验证触发器的行为。

```
SQL> delete from emp
  2  where ename ='FORD';
deleting........

已删除 1 行。
```

从上述输出可以看出，当删除表 EMP 中的记录之前，触发器将执行触发事件并向标准输出打印一行提示，然后删除满足删除条件的行记录。

8.1.2　创建基于 Java 语言的触发器

Oracle 允许在创建触发器时，使用 Java 代码执行触发器的执行部分，但是需要读者注意，创

建基于 Java 语言的触发器不是直接将 Java 代码关联为触发器，而是在触发器的执行部分调用 Java 代码。下面我们创建一个包含 Java 代码的触发器，它的作用是当对 EMP 表执行删除操作前，将用户名、删除时间和删除行为保存到表 USER_MODIFY_TABLE 中。首先我们创建一个 Java 类 DelempTrigger，如实例 8-4 所示。

实例 8-4　创建 Java 类 DelempTrigger。

```
import java.io.*;
import java.sql.*;
import oracle.sql.*
import oracle.core.lmx.*

public class DelempTrigger{

public static void recordDeleteData() throws SQLException,CoreException
 {
    Connection conn = JDBCConnection.defaultConnection();
    Statement stm = conn.CreateStatement();
    stm.execute("INSERT INTO user_modify_table
               VALUES (user,sysdate,'deleting') ");
    stm.close();
    return;
 }
}
```

该类定义了一个 Public Static 方法 recordDeleteData()，它是 Java 代码执行触发器主体行为的方法，然后编译并生成 DelempTrigger.class 文件。



在编译该 DelempTrigger .java 文件前需要在环境变量中将 Oracle 自带的一个 class12.jar 文件包含在内。

创建一个过程用于包装 Java 代码，如实例 8-5 所示。

实例 8-5　创建包含 Java 代码的存储过程 record_deleteemp_user_trigger。

```
Create or replace procedure record_deleteemp_user_trigger( )
Is language java
Name 'DelempTrigger. recordDeleteData( ) ';
```

最后我们定义一个存储过程，在过程的执行部分调用封装了 Java 代码的存储过程，如实例 8-6 所示。

实例 8-6　创建基于 Java 语言的存储过程 user_change_empdata。

```
CREATE OR REPLACE TRIGGER user_change_empdata
BEFORE delete ON emp
FOR EACH ROW
BEGIN
--调用封装了 JAVA 代码的存储过程
 call record_deleteemp_user_trigger();
END user_change_empdata;
```

在该存储过程的执行部分调用了另一个存储过程 record_deleteemp_user_trigger，而该过程是对

Java 代码的封装。

8.2 触发器的分类

Oracle 的数据库触发器可以对不同的操作类型实现某种行为，如审计或安全考虑等，这些操作包括 DML 操作、DDL 操作和数据库级操作。

1. 基于 DML 操作的触发器

触发器可以在当用户对一个表的 INSERT、UPDATE 和 DELETE 操作时触发行为，也可以实现对表的每一行进行 DML 操作时，实现触发器行为，此时需要在触发器的定义中使用 FOR EACH ROW 语句来说明操作的每一行都触发哪种触发器行为。

创建该类触发器的语法格式如下所示。

```
CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE|AFTER]
[INSERT|UPDATE|DELETE] ON table_name [FOR EACH ROW [WHEN cond]]
Trigger_SQL_Statements;
```

创建触发器时 OR REPLACE 语句可以使用，也可以不使用，使用它的目的是：如果在当前用户模式下已经创建了该触发器，则覆盖原触发器；BEFORE|AFTER 说明触发器被触发的时机，而 INSERT|UPDATE|DELETE 说明触发器被触发的事件，如 BEFORE INSERT 说明在向表中插入数据前激发触发器行为；ON table_name 说明触发器作用的表名；FOR EACH ROW 说明触发器对操作（如 INSERT）涉及的每一行都激发触发器行为，这样的触发器称为行级触发器；WHEN cond 给出了更具体的条件，当满足某种条件时，再执行触发器行为。

2. 基于 DDL 操作的触发器

DDL 操作如 CREATE、ALTER 和 DROP，在执行这些操作前或者之后实现触发器行为，如用户删除了一个表，此时需要一个触发器来记录该用户删除的表的信息以及该用户名，以作为用户操作日志，这也是此类触发器的典型应用。

创建该类触发器的语法格式如下所示。

```
CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE|AFTER]
[CREATE|ALTER|DROP] ON database_name [WHEN cond]]
Trigger_SQL_Statements;
```

此类触发器在数据库中创建、删除数据库对象或者执行 ALTER 指令时激发触发器行为。

3. 基于数据库级操作的触发器

数据库级操作是指 START、SHUTDOWN、LOGON、LOGOFF 等与数据库相关的操作，如用户登录时记录该用户登录的时间和用户名，而当用户退出时，也记录该用户的退出时间等。对于数据库启动 START 和数据库关闭 SHUTDOWN 等，同样可以编写符合业务需求的触发器。

创建该类触发器的语法格式如下所示。

```
CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE|AFTER]
[START|SHUTDOWN|LOGON|LOGOFF] ON database_name [WHEN cond]]
Trigger_SQL_Statements;
```

此类触发器在数据库级行为，如关闭和启动数据库、用户登录和退出数据库时激发触发器行为。

按照触发器操作对象的粒度不同，也可以分为语句级触发器和行级触发器，但是使用操作类型分类更加直观。

8.3　触发器的权限

在用户创建触发器时，必须具有 CREATE TRIGGER 权限，如果用户不具备这个权限，则需要在 DBA 用户下，使用 GRANT CREATE TRIGGER TO user_name 指令赋予当前用户权限 ，而如果需要在当前用户下，创建其他用户的触发器，则需要具有 CREATE ANY TRIGGER 的权限，这里 ANY 的含义就是为任何用户创建触发器。如果要创建的触发器是作用在数据库上的，如对 START 或 SHUTDOWN 事件激发触发器，则需要具有 ADMINISTER DATABASE TRIGGER 的系统权限。

例如在 SCOTT 模式下，可以通过如下方式查看当前用户是否具有创建触发器的权限，如实例 8-7 所示。

实例 8-7　查看当前用户的系统权限。

```
SQL> conn system/oracle@orcl
已连接。
SQL> select *
  2  from dba_sys_privs
  3  where grantee = 'SCOTT';

GRANTEE                    PRIVILEGE                    ADM
---------- ---------------------------------------- ------------
SCOTT                       UNLIMITED TABLESPACE         NO
SCOTT                       CREATE TRIGGER               NO
```

从输出可以看出，该 SCOTT 用户具有创建触发器的权利，但是只能创建自己模式下的触发器，如果 PRIVILEGE 为 CREATE ANY TRIGGER，则可以创建任何模式下的触发器，可以通过如下授权实现，代码如下所示。

```
SQL> GRANT create any trigger to SCOTT;

授权成功。
```

此时，显示授权成功，为了验证用户 SCOTT 的权限信息，再次使用数据字典 DBA_SYS_PRIVES 来查看用户 SCOTT 的系统权限，如实例 8-8 所示。

实例 8-8　查看用户是否具有 CREATE ANY TRIGGER 的权利。

```
SQL> col grantee for a15
SQL> col privilege for a20
SQL> col admin_option for a15
SQL> select *
  2  from dba_sys_privs
  3  where grantee = 'SCOTT'
  4* and privilege like 'CREATE%'
```

```
GRANTEE          PRIVILEGE              ADMIN_OPTION
------- -------------------------------- --------------------------
SCOTT            CREATE TRIGGER         NO
SCOTT            CREATE ANY TRIGGER     NO
```

此时，数据字典 DBA_SYS_PRIVS 中记录了用户 SCOTT 具有 CREATE ANY TRIGGER 的权限。



ADMIN_OPTION 选项的含义是 SCOTT 用户具有的权利是否可以再赋予其他用户：如果 ADMIN_OPTION 为 NO，则说明对应的权利不能再赋予其他用户；如果 ADMIN_OPTION 为 YES，则说明对应的权利可以继续赋予其他用户。

8.4　触发器中的新值和旧值

在创建基于 DML 操作的触发器时，由于操作的是表对象，所以有一个可选项，即 FOR EACH ROW，以实现对每一行都激发触发器行为。此时 Oracle 提供了两个临时表来访问每行中的新值和旧值，即:new 和:old。下面我们给出实例来说明它们的作用。

编写一个触发器，作用是更新 SCOTT 用户的表 EMP 中的记录后，再将更新的行记录对象的工资的旧值和新值打印到标准输出装置，如实例 8-9 所示。

实例 8-9　创建表 EMP 的 UPDATE 触发器。

```
create or replace trigger update_emp_trigger
 after update on emp
 for each row
begin
dbms_output.put_line('old value is '||:old.sal);
dbms_output.put_line('new value is'||:new.sal);
end update_emp_trigger;
```

我们将该文件保存在 F 盘根目录下，名字为 updateemp.sql，然后执行该脚本文件，创建触发器 UPDATE_EMP_TRIGGER，代码如下所示。

```
SQL> @f:\updateemp;
  8  /

触发器已创建
```

此时，已经成功创建触发器 UPDATE_EMP_TRIGGER，通过数据字典查看该触发器的信息，如实例 8-10 所示。

实例 8-10　通过数据字典 USER_OBJECTS 查看触发器信息。

```
SQL> col object_name for a25
SQL> select object_name,object_type,created,status
  2  from user_objects
  3  where object_type = 'TRIGGER'
  4* and status = 'VALID'

OBJECT_NAME               OBJECT_TYPE        CREATED        STATUS
```

```
------ ------------------------------- ----------------------- -------------
CHECK_EMP_SALARY          TRIGGER            20-8 月 -09     VALID
UPDATE_EMP_TRIGGER        TRIGGER            19-8 月 -09     VALID
```

由此可见，触发器 UPDATE_EMP_TRIGGER 创建成功并记录在数据字典中，该触发器是有效的触发器（STATUS 为 VALID）。下面我们更新表 EMP 中的数据，以便观察触发器的行为，如实例 8-11 所示。

实例 8-11　更新表 EMP 中的数据。

```
SQL> set serveroutput on
SQL> update emp
  2  set sal = sal+200
  3  where job ='CLERK';
old value is 800
new value is 1000
old value is 1100
new value is 1300
old value is 950
new value is 1150
old value is 1300
new value is 1500

已更新 4 行。
```

从输出可以看出，在表 EMP 上更新相关列 SAL 的值之后，触发器开始工作：先输出每一行中 SAL 的旧值，再输出每一行中更新后的 SAL 的新值，因为我们的更新语句是将 EMP 表中岗位是 CLERK 的员工工资统一增加 200，所以每行的旧值总是比新值少 200。因为更新了 4 行记录，所以触发器的输出有 8 行，每行涉及一对新值和旧值。

8.5　审核触发器的创建

本节将给出一个具有实际意义的触发器，即审核触发器，当用户操作一个重要的表时，如插入数据和更新数据，我们希望能够记录该用户的名称和更改时间等信息，以备将来审核时使用。

在创建审核触发器之前，我们需要先创建一个表，该表用于存储审核信息，如实例 8-12 所示。

实例 8-12　创建审核表。

```
SQL> create table user_modify_table
  2  ( username  varchar2(20),
  3    modifytime   date,
  4    content   varchar2(20));

表已创建。
```

成功创建表 USER_MODIFY_TABLE 之后，可使用 DESC 指令查看该表的结构信息，代码如下所示。

```
SQL> desc user_modify_table;
 名称                                       是否为空? 类型
 ------------------------- ------------ ------------------------
 USERNAME                                            VARCHAR2(20)
```

```
    MODIFYTIME                                        DATE
    CONTENT                                           VARCHAR2(20)
```

此时，我们已做好准备，即使用表 USER_MODIFY_TABLE 存储用户的信息，下面我们创建审核触发器，当发生对表 EMP 的 INSERT 操作和 UPDAT 操作时，将用户名、操作日期和操作内容记录到表 USER_MODIFY_TABLE 中，如实例 8-13 所示。

实例 8-13　创建审核触发器。

```
CREATE OR REPLACE TRIGGER user_change_empdata
BEFORE update or insert ON emp
FOR EACH ROW
BEGIN
    IF inserting THEN
INSERT INTO user_modify_table
 VALUES (user,sysdate,'inserting');
END IF;
IF updating THEN
INSERT INTO user_modify_table
 VALUES (user,sysdate,'updating');
END IF;
END user_change_empdata;
/
```

将上述代码在记事本中编辑并保存在 F 盘的根目录下，文件名为 user_change_empdata.sql，下面执行该脚本文件，创建触发器，代码如下所示。

```
SQL> @f:\user_change_empdata.sql;

触发器已创建
```

同样，查询该触发器在数据字典 USER_OBJECTS 中的信息，如实例 8-14 所示。

实例 8-14　查询触发器 USER_CHANGE_EMPDATA 的信息。

```
SQL> select object_name,object_id,object_type,created,status
  2  from user_objects
  3  where object_type = 'TRIGGER'
  4  and object_name like 'USER%';

OBJECT_NAME           OBJECT_ID OBJECT_TYPE    CREATED    STATUS
- -------------- ----------------------- -------------------- --------------
USER_CHANGE_EMPDATA  53299     TRIGGER        20-8 月 -09   VALID
```

由此可见，已经成功创建了触发器 USER_CHANGE_EMPDATA，而且该触发器是有效（VALID）的，所以当对表 EMP 进行 UPDATE 或 INSERT 操作时会激发触发器，将用户操作信息记录到表 USER_MODIFY_TABLE 中，进行 INSERT 和 UPDATE 操作，如实例 8-15 所示。

实例 8-15　对表 EMP 进行 INSERT 和 UPDATE 操作。

```
SQL> insert into emp (empno,ename,sal,job)
  2  values (7899,'tom',10000,'MANAGER');

已创建 1 行。
SQL> update emp
  2  set sal = sal +200
```

```
  3  where job ='CLERK';

已更新 4 行。
```

为了验证触发器是否被触发，可以查看表 USER_MODIFY_TABLE 的内容，如实例 8-16 所示。

实例 8-16　查看表 USER_MODIFY_TABLE。

```
SQL> SELECT *
  2  FROM USER_MODIFY_TABLE;

USERNAME         MODIFYTIME     CONTENT
------------- ------------------------- -----------------
SCOTT              20-8 月 -09      inserting
SCOTT              20-8 月 -09      updating
SCOTT              20-8 月 -09      updating
SCOTT              20-8 月 -09      updating
SCOTT              20-8 月 -09      updating
```

由此可见，对表 EMP 的一行进行了 INSERT 操作，对表中的 4 行进行了 UPDATE 操作。从表 USER_MODIFY_TABLE 的记录可以知道触发器 USER_CHANGE_EMPDATA 已被成功触发。

8.6　删除触发器的创建

删除触发器的作用是对表中的数据进行删除 DELETE 操作时记录删除的内容，在某些场合中是出于备份的需要，同样我们使用表 EMP 作为删除触发器的操作对象。

为了保存删除的数据，我们需要创建一个记录删除数据的表，该表的结构与 EMP 表的结构对应，即列的数据类型和数量都相等，如实例 8-17 所示。

实例 8-17　创建记录删除数据的表 BACKUP_DELETE_EMP_TABLE。

```
SQL> create table backup_delete_emp_table
  2  (backempno  number(4),
  3  backename   varchar2(10),
  4  backjob     varchar2(9),
  5  backmgr    number(4),
  6  backhiredate date,
  7  backsal    number(7,2),
  8  backcomm   number(7,2),
  9  backdeptno number(2));

表已创建。
```

由结果可知，可成功创建记录删除数据的表 BACKUP_DELETE_EMP_TABLE。下面开始创建一个删除触发器，用于备份删除的表 EMP 中的记录，如实例 8-18 所示。

实例 8-18　创建删除触发器。

```
create or replace trigger backup_emp_trigger
before delete on emp
for each row
begin
insert into backup_delete_emp_table
```

```
values(:old.empno,:old.ename,:old.job,:old.mgr,:old.hiredate,
      :old.sal,:old.comm,:old.deptno);
end backup_emp_trigger;
```

将上述文件在记事本中编辑，命名为 backup_emp_trigger.sql 并保存在 F 盘根目录下，下面执行脚本文件，从而创建触发器 backup_emp_trigger.sql，代码如下所示。

```
SQL> @f:\backup_emp_trigger.sql;
  9  /

触发器已创建
```

输出提示触发器已创建成功，为了演示触发器工作的结果，我们先在 EMP 表上执行两个删除操作，如实例 8-19 所示。

实例 8-19　在 EMP 表执行删除操作。

```
SQL> delete from emp
  2  where ename = 'tom';

已删除 1 行。
SQL> delete from emp
  2  where ename = 'SCOTT';

已删除 1 行。
```

因为触发器 backup_emp_trigger 是作用在表 EMP 上的，并且当从表 EMP 删除数据记录时，激发该触发器，使得被删除的记录保存在表 BACKUP_DELETE_EMP_TABLE 中。下面我们查询表 BACKUP_DELETE_EMP_TABLE 的内容，如实例 8-20 所示。

实例 8-20　查询表 BACKUP_DELETE_EMP_TABLE 的内容。

```
SQL> select backempno,backename,backjob,backhiredate,backsal
  2  from backup_delete_emp_table;

 BACKEMPNO BACKENAME  BACKJOB   BACKHIREDATE      BACKSAL
- - -------------- -------------------------------- ----------------
     7899     tom           MANAGER                        10000
     7788     SCOTT       ANALYST    19-4 月 -87             3000
```

可以看到员工 tom 和 SCOTT 正是我们刚刚删除的记录，而他们的记录信息保存在表 BACKUP_DELETE_EMP_TABLE 中，说明触发器 backup_emp_trigger 被 DELETE 事件成功激发。

8.7　触发器的条件语句

在触发器中为了更加细粒度地控制触发器的激发条件，所以允许使用条件语句，主要有两类条件语句：一个是 WHEN 子句，另一个 IF 子句。本节将依次讲解这两类条件语句的用法，并通过实例使得读者理解如何运用这两类条件语句。

8.7.1　WHEN 条件语句

对于创建的触发器，往往需要进行更详细的条件控制，使用 WHEN 子句可以设置基于行级的触发器条件，即对表的每一行的操作进行更加细致的条件判断，从而执行某些行为。在触发器中使用 WHEN 子句的语句结构如下所示。

```
CREATE OR REPLACE TRIGGER when_emp_trigger
BEFORE UPDATE OR INSERT ON emp
FOR EACH ROW
WHEN (condition)
BEGIN
Trigger_body;
END when_emp_trigger;
```

下面更改删除触发器的实例，在删除表 EMP 的一行记录时，先判断要删除的岗位，即 JOB，如果岗位是 MANAGER 或者 PRESIDENT，则备份删除的记录，若删除的是其他记录，则不需要做备份。修改后的删除触发器如实例 8-21 所示。

实例 8-21　创建使用 WHEN 子句的删除触发器。

```
CREATE OR REPLACE TRIGGER when_backup_emp_trigger
BEFORE delete ON emp
FOR EACH ROW
WHEN (old.job IN 'MANAGER,PRESIDENT')
BEGIN
INSERT INTO backup_delete_emp_table
VALUES(:old.empno,:old.ename,:old.job,:old.mgr,:old.hiredate,
      :old.sal,:old.comm,:old.deptno);
END when_backup_emp_trigger;
```

此时在 FOR EACH ROW 后使用了 WHEN 条件语句，这样使得触发器触发的时机条件更加详细，即实现更加细粒度的条件控制。



在使用 WHEN 子句时，WHEN 子句中的 new 或者 old 不使用:old 或:new 的形式，并且在 WHEN 子句后没有分号。

8.7.2　IF 条件语句

在对表创建触发器时，可通过几个 DML 操作来激发触发器行为，这些操作包括 INSERT、UPDATE 和 DELETE，一个触发器可以响应多个触发事件，而执行不同的行为，如被 INSERT 事件触发时，希望保存旧的行记录，并保存该操作事件，而使用 DELETE 事件触发时，则需要记录用户名和 DELETE 事件，此时就需要一个条件判断。

Oracle 设计了三个布尔表达式来表示三个不同的触发事件，即 INSERT、UPDATE 和 DELETE，其使用方式如下所示。

```
IF [INSERT | UPDATE | DELETE ] THEN
  trigger_body;
END IF ;
```

8.8 触发器的管理

触发器的管理是触发器维护的重要内容，首先在维护前需要知道触发器的信息，Oracle 提供了 USER_TRIGGERS 数据字典来查看当前用户的触发器信息。触发器由于依赖性无效而失效，可以重新编译该触发器，如果临时不需要执行触发器可以将其屏蔽掉，如果永久不需要该触发器可以删除触发器。

8.8.1 查看触发器

本章已经建立了很多触发器，都是对表 EMP 的事件触发行为，那么如何查看我们创建的触发器的信息呢？Oracle 提供了数据字典 USER_TRIGGERS。通过它可以查看触发器的所有信息。首先通过实例 8-22 查看数据字典的结构。

实例 8-22 查看数据字典 USER_TRIGGERS 的结构。

```
SQL> desc user_triggers;
 名称                              是否为空? 类型
 ---------------------------------- -------- ------------------
 TRIGGER_NAME                               VARCHAR2(30)
 TRIGGER_TYPE                               VARCHAR2(16)
 TRIGGERING_EVENT                           VARCHAR2(227)
 TABLE_OWNER                                VARCHAR2(30)
 BASE_OBJECT_TYPE                           VARCHAR2(16)
 TABLE_NAME                                 VARCHAR2(30)
 COLUMN_NAME                                VARCHAR2(4000)
REFERENCING_NAMES                           VARCHAR2(128)
WHEN_CLAUSE                                 VARCHAR2(4000)
 STATUS                                     VARCHAR2(8)
 DESCRIPTION                                VARCHAR2(4000)
 ACTION_TYPE                                VARCHAR2(11)
 TRIGGER_BODY                               LONG
```

下面我们依次说明这些参数的作用，这样读者就可以通过数据字典 USER_TRIGGERS 查询需要的当前用户的触发器信息。

- TRIGGER_NAME：触发器名称，如 BACKUP_EMP_TRIGGER。
- TRIGGER_TYPE：触发器类型，说明是行级触发器还是语句级触发器，如果是行级触发器，则该值为 BEFORE EACH ROW 或 AFTER EACH ROW。
- TRIGGERING_EVENT：触发事件，如 DELETE、UPDATE、INSERT 等。
- TABLE_OWNER：触发器所关联的表的拥有者。
- BASE_OBJECT_TYPE：触发器所关联的是表 TABLE，还是数据库 DATABASE。
- TABLE_NAME：触发器所关联的表名。
- COLUMN_NAME：触发器所关联的列名，Oracle 允许更细粒度的触发器激发控制，可以在用户操作表的某一列时执行触发器行为。
- WHEN_CLAUSE：触发器中的 WHEN 条件语句内容。
- STATUS：说明当前的触发器是否被屏蔽，DISABLED 表示被屏蔽了，即没有激活该触

发器，ENABLED 表示激活了该触发器，可以使用。

- DESCRIPTION：简单说明触发器名称、触发事件、触发时机，以及关联的对象和触发器类型（行级触发器还是语句级触发器），一个 DESCRIPTION 的示例如下。

```
backup_emp_trigger
before delete on emp
for each row
```

- TRIGGER_BODY：触发器的执行部分。触发器 BACKUP_EMP_TRIGGER 的 TRIGGER_BODY 内容如下所示。

```
begin
insert into backup_delete_emp_
table
values(:old.empno,:old.ename,:
old.job,
```

下面，我们给出一个查看触发器 BACKUP_EMP_TRIGGER 信息的实例，如实例 8-23 所示。

实例 8-23　查看触发器 BACKUP_EMP_TRIGGER 的信息。

```
SQL> set line 120
SQL> col status for a15
SQL> col triggering_event for a15
SQL> col description for a25
SQL>select trigger_name,triggering_event,description,status
  2  from user_triggers
  3* where trigger_name = 'BACKUP_EMP_TRIGGER'

TRIGGER_NAME       TRIGGERING_EVEN DESCRIPTION         STATUS
--------------- -------- --------------------------------- --------------
BACKUP_EMP_TRIGGER   DELETE          backup_emp_trigger     ENABLED
                                      before delete on emp
                                      for each row
```

可见触发器 BACKUP_EMP_TRIGGER 的触发事件是 DELETE 操作，作用的表为 EMP，当前的状态为激活状态，因为 STATUS 为 ENABLED。

8.8.2　重新编译触发器

当触发器被标记为无效时，此时它不能被执行，需要重新编译该触发器，手工编译触发器的指令如下所示。

```
ALTER TRIGGER trigger_name COMPILE;
```

下面给出实例 8-24，用于说明如何重新编译触发器 BACKUP_EMP_TRIGGER。

实例 8-24　手工编译触发器 BACKUP_EMP_TRIGGER。

```
SQL> alter trigger backup_emp_trigger compile;

触发器已更改
```

8.8.3 屏蔽触发器

如果一个触发器不需要执行，但是又不希望删除它，则可以屏蔽该触发器，屏蔽触发器可使用指令 ALTER TRIGGER trigger_name DISABLE 实现，如实例 8-25 所示。

实例 8-25 屏蔽触发器 BACKUP_EMP_TRIGGER。

```
SQL> alter trigger backup_emp_trigger disable;

触发器已更改
```

在屏蔽触发器后，我们通过实例查看、开启触发器 BACKUP_EMP_TRIGGER 的状态，如实例 8-26、实例 8-27 所示。

实例 8-26 查看触发器 BACKUP_EMP_TRIGGER 的状态。

```
SQL> col table_name for a10
SQL>select trigger_name,triggering_event,table_name,status
  2  from user_triggers
  3* where trigger_name ='BACKUP_EMP_TRIGGER'

TRIGGER_NAME          TRIGGERING_EVEN TABLE_NAME STATUS
----------------- --------- -------------------- ---------------
BACKUP_EMP_TRIGGER   DELETE           EMP         DISABLED
```

由此可见，触发器 BACKUP_EMP_TRIGGER 已经被屏蔽了，STATUS 的值为 DISABLED，那么如何激活该触发器呢？启动触发器的语句实现如下。

```
ALTER TRIGGER trigger_name ENABLE;
```

实例 8-27 开启触发器 BACKUP_EMP_TRIGGER。

```
SQL> alter trigger backup_emp_trigger enable;

触发器已更改
```

下面通过数据字典 USER_TRIGGERS 查看触发器 BACKUP_EMP_TRIGGER 的状态，如实例 8-28 所示。

实例 8-28 查看开启后的触发器 BACKUP_EMP_TRIGGER 的状态。

```
SQL> select trigger_name,triggering_event,table_name,status
  2  from user_triggers
  3  where trigger_name ='BACKUP_EMP_TRIGGER';

TRIGGER_NAME          TRIGGERING_EVEN TABLE_NAME STATUS
----------------- --------- -------------------- ---------------
BACKUP_EMP_TRIGGER   DELETE          EMP        ENABLED
```

注意此时触发器 BACKUP_EMP_TRIGGER 的 STATUS 为 ENABLED，说明该触发器被重新启动。

8.8.4　删除触发器

当不需要一个已经创建的触发器时，可以删除该触发器，其指令如下所示。

```
DROP TRIGGER trigger_name
```

下面我们给出实例 8-29，用于删除触发器 BACKUP_EMP_TRIGGER。

实例 8-29　删除触发器 BACKUP_EMP_TRIGGER。

```
SQL> drop trigger backup_emp_trigger;

触发器已删除。
```

为了验证是否成功删除触发器 BACKUP_EMP_TRIGGER，再通过数据字典 USER_TRIGGERS 来查看该触发器的信息，如实例 8-30 所示。

实例 8-30　查看触发器 BACKUP_EMP_TRIGGER 是否存在。

```
SQL> select trigger_name,triggering_event,status
  2  from user_triggers
  3  where trigger_name = 'BACKUP_EMP_TRIGGER';

未选定行
```

输出结果显示“未选定行”，说明数据字典中没有该触发器的记录，成功删除触发器 BACKUP_EMP_TRIGGER。

8.9　本章小结

本章介绍了数据库中非常重要的对象——触发器，触发器允许 Oracle 基于业务规则或数据库安全的考虑进行编码。触发器是使用 PL/SQL 语言编写的代码块，Oracle 允许使用触发器激发基于表和数据库的行为，如对表进行 DML 操作时，使用触发器记录用户的操作轨迹，当数据库启动或关闭时记录数据库的状态信息等。

本章在开始部分介绍了如何创建触发器，读者通过它可以对触发器建立感性认识。笔者基于操作类型对触发器进行了分类。在实际操作中审核触发器和删除触发器是具有现实意义的，略加修改就可以创建出符合自己业务需求的此类触发器。在触发器中允许使用条件语句进行触发器事件的细粒度控制，如 WHEN 子句和 IF 子句。而触发器管理是触发器维护的重要内容，在本章的最后介绍了如何使用数据字典查看触发器的内容，以及重新编译触发器、屏蔽触发器和删除触发器。

第 9 章
◂ 存储过程 ▸

存储过程是 Oracle 数据库的一种数据库对象，它存储在数据库的服务器端，执行用户的逻辑计算，存储过程在数据库服务器端存储并且运行，一次调用后可以反复地在内存中使用，存储过程使用 PL/SQL 语言、Java 语言来编写，存储过程被显式地调用，以完成过程定义的计算任务。存储过程可以接收各种 Oracle 定义的参数，用户可以在 SQL*Plus 或者应用程序中调用 PL/SQL 过程，一旦过程被创建，则在数据字典中记录该数据库的对象信息，其数据库对象类型为 Procedure。

9.1 存储过程的结构

存储过程是保存在数据库服务器上的程序单元，这些程序单元对数据库的重复操作作用很大。一个存储过程由三部分组成，即声明区、子程序区和异常处理区，其组成结构如图 9-1 所示。

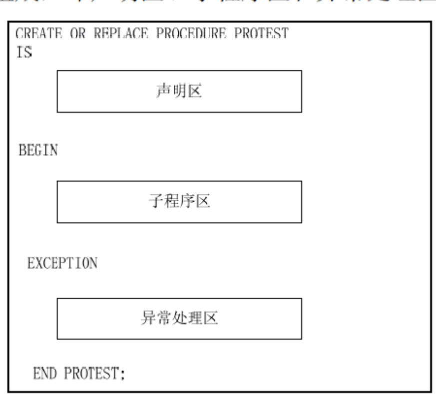


图 9-1 存储过程的组成图

1．声明区

声明区位于 PROCEDURE 和 BEGIN 之间，在声明区中可定义变量，代码如下所示。

```
CREATE OR REPLACE PROCEDURE PROTEST
IS
--声明变量
   xxx number ;
   yyy varchar2(20) :='oracle';
--声明 REF 游标
```

```
type empcursor is ref cursor;
--异常对象
read_disk_refused exception;
--内嵌函数或其他存储过程。
FUNCTION foo RETURN BOOLEAN IS
BEGIN
RETURN (XXX>1000);
END IF;
BEGIN
……
END;
```

在声明区中，声明了三个变量：一个为 NUMBER 类型，一个为 VARCHAR2 类型，最后一个为 REF 游标，还声明了一个内嵌函数。

2. 子程序区

子程序区即 BEGIN…END 之间的部分，这部分包括 PL/SQL 代码逻辑，代码逻辑对于声明区而言是可见的，这部分是存储过程执行其功能的主体部分，并且在过程的定义中，子程序部分是不可缺少的，要求在 BEGIN…END 之间至少存在一条 PL/SQL 语句，可以是 NULL，示例代码如下所示。

```
CREATE OR REPLACE PROCEDURE PROTEST
IS
BEGIN
--此处是 PL/SQL 代码逻辑
logical statement;
END;
```

3. 异常处理区

异常处理区处理在子程序执行中发生的异常，示例代码如下所示。

```
CREATE OR REPLACE PROCEDURE PRO(employee_age,IN NUMBER)
IS
    age    NUMBER;
    mydate DATE;
    dateexp EXCEPTION;
 BEGIN
   mydate := 'TIME_STRING';
   age := employee_age;
   IF age >150
THEN
RAISE dateexp;
  EXCEPTION
   WHEN dateexp THEN
      handling dateexp;
   WHEN OTHERS   THEN
      handling other exps;
END;
```

在上例中，我们给出了一个异常 dateexp，该异常在过程的声明区中声明，当 age>150 时就抛出 dateexp 异常，在异常处理区 Exception 后进行处理，注意 WHEN 子句对应了异常类型，如果是其他类型，则默认在 WHEN OTHERS THEN 后处理。

9.2　存储过程的初体验

在学习了存储过程的结构之后，开始尝试创建一个存储过程，该过程的目的是更新 SCOTT 用户的 EMP 表中的员工薪水，满足条件的员工为工资低于平均工资的员工，将这些员工的工资在原有基础上统一增加 15%。下面是过程代码，如实例 9-1 所示。

实例 9-1　创建存储过程 IncreSal。

```
SQL> create or replace procedure IncreSal
  2  is
  3    var_average number;
  4    cursor cur_emp is
  5    select ename,sal from emp;
  6  begin
  7    select avg(sal) into var_average from emp;
  8    for cursor_emp in cur_emp
  9     loop
 10       update emp
 11          set sal=sal*1.15
 12            where sal<var_average;
 13       dbms_output.put_line(cursor_emp.ename||' updated!');
 14     end loop;
 15*  end;
SQL> /

Procedure created.
```

在上例中，我们使用连接 1 创建了过程 IncreSal，然后在执行该过程之前，先查询当前所有小于平均工资的员工信息，这样就可以与执行过程后的数据进行对比，从中可以发现一些问题。现在我们查询当前表 emp 的员工平均工资，此时新建一个连接，称为连接 2，如实例 9-2 所示。

实例 9-2　新建连接 2，并查询表 EMP 的员工平均工资。

```
[oracle@localhost ~]$ sqlplus scott/oracle

SQL*Plus: Release 11.2.0.1.0 Production on Mon Mar 12 15:05:02 2012

Copyright (c) 982, 2009, Oracle.  All rights reserved.


Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> show user
USER is "SCOTT"
SQL> select avg(sal) from emp;

  AVG(SAL)
----------
3063.85929.
```

这里我们选择使用新建连接，而不是在创建过程后直接在 SYS 用户下执行，这样做是为了后

面的对比。下面我们在连接 1 中的 SCOTT 用户下执行该游标，如实例 9-3 所示。

实例 9-3　在连接 1 中执行过程 IncreSal。

```
SQL> exec IncreSal;
SMITH updated!
ALLEN updated!
WARD updated!
JONES updated!
MARTIN updated!
BLAKE updated!
CLARK updated!
SCOTT updated!
KING updated!
TURNER updated!
ADAMS updated!
JAMES updated!
FORD updated!
MILLER updated!

PL/SQL procedure successfully completed.
```

我们看到有 14 个员工的工资被更新，下面我们在当前会话下查询 EMP 表的平均工资数额，如实例 9-4 所示。

实例 9-4　查询 EMP 表的平均工资数额。

```
SQL> show user
USER is "SYS"
SQL> select avg(sal) from scott.emp;

  AVG(SAL)
----------
3469.86786
```

从输出可以知道，已经更新了所有低于平均工资的员工工资，但是注意这是在当前会话中完成的，我们继续在连接 2 的会话中查询表 EMP 中的员工平均工资，如实例 9-5 所示。

实例 9-5　在连接 2 的会话中查询表 EMP 中的员工平均工资。

```
SQL> select avg(sal) from emp;

  AVG(SAL)
----------
3063.85929.
```

此时，我们会发现平均工资依然没有变化，这个数值间接地告诉我们：虽然在连接 1 中执行过程 IcreSal 来更新员工薪水，但是并没有提交更改，所以在连接 2 中看不到变化。

如果要完成该过程，即提交数据的目的，必须添加 COMMIT 关键字，可以在 END LOOP 之前或者之后添加，如果在之前添加就会每次更改一条记录并提交一次，如果在 END LOOP 之后添加，则在更该后批量提交，显然后者的效率更高。修改代码如实例 9-6 所示。

实例 9-6　修改过程 IncreSal 的定义。

```
create or replace procedure IncreSal
```

```
is
  var_average number;
  cursor cur_emp is
   select ename,sal from emp;
begin
  select avg(sal) into var_average from emp;
  for cursor_emp in cur_emp
   loop
     update scott.emp
        set sal=sal*1.15
          where sal<var_average;
     dbms_output.put_line(cursor_emp.ename||' updated!');
   end loop;
   commit;
 end;
/
```

读者可以自行修改过程 IncreSal 的代码，在 END LOOP 之后添加 COMMIT 关键字，并验证更改的提交效果。

9.3　存储过程的信息和定义查询

通过上节的操作，我们在 SCOTT 模式下创建了过程 IncreSal。过程是数据库的一种对象类型，这样的对象显然会记录在数据字典中，可以通过 user_objects 查询该对象的信息，如过程名、过程状态等，查询过程 IncreSal 的信息，如实例 9-7 所示。

实例 9-7　查询过程 IncreSal 在数据字典中的定义。

```
SQL> select object_id,object_type,status,created from user_objects
  2  where object_name='INCRESAL';

 OBJECT_ID OBJECT_TYPE        STATUS  CREATED
----------------- -------------- ------------- ---------------
     75130  PROCEDURE          VALID   12-MAR-12
```

从上述输出可以看出过程 IncreSal 的 ID、状态以及创建时间。如果想知道该过程是如何创建的，应该怎样操作呢？Oracle 提供了 DBMS_METADATA 包，该包的过程 get_dll 可以获得数据库对象的创建过程，如实例 9-8 所示，我们使用该包来获得过程 IncreSal 的定义。

实例 9-8　获得过程 IncreSal 的定义。

```
SQL> set pagesize 1000;
SQL> set long 1000
SQL>  select dbms_metadata.get_ddl('PROCEDURE','INCRESAL') FROM DUAL;

DBMS_METADATA.GET_DDL('PROCEDURE','INCRESAL')
-------------------------------------------------------------
  CREATE OR REPLACE PROCEDURE "SCOTT"."INCRESAL"
is
  var_average number;
  cursor cur_emp is
   select ename,sal from emp;
begin
```

```
  select avg(sal) into var_average from emp;
  for cursor_emp in cur_emp
   loop
     update scott.emp
        set sal=sal*1.15
          where sal<var_average;
     dbms_output.put_line(cursor_emp.ename||' updated!');
   end loop;
 end;
```

DBMS_METADATA 包可以获得所有数据库对象的定义信息，如果读者在维护数据库时发现一个设计良好的存储过程，就可以通过这个包获得一个优秀的存储过程，从中获得更好地启发。此外，也可以使用数据字典来获得对象定义，该数据字典为 user_source。下面是该数据字典的结构，如实例 9-9 所示。

实例 9-9　数据字典 user_source 的结构。

```
SQL> desc user_source;
 Name                                       Null?    Type
 ---------------------- -------- ----------------------------
 NAME                                                VARCHAR2(30)
 TYPE                                                VARCHAR2(12)
 LINE                                                NUMBER
 TEXT                                                VARCHAR2(4000)
```

从定义中可以看出属性 TEXT 为 4000 个字符，用来存储对象的大文本定义，类型 TYPE 包括表、触发器、存储过程等。下面我们通过该数据字典获取过程 IncreSal 的定义，如实例 9-10 所示。

实例 9-10　通过该数据字典获取过程 IncreSal 的定义。

```
SQL> select text from user_source where name='INCRESAL';

TEXT
------------------------------------------------------------
procedure IncreSal
is
  var_average number;
  cursor cur_emp is
   select ename,sal from emp;
begin
  select avg(sal) into var_average from emp;
  for cursor_emp in cur_emp
   loop
     update scott.emp
        set sal=sal*1.15
          where sal<var_average;
     dbms_output.put_line(cursor_emp.ename||' updated!');
   end loop;
 end;

15 rows selected.
```

可以看到，使用该数据字典与使用工具包获得的定义效果相同，其实包 DBMS_METADATA 就是从数据字典中获得数据库对象的定义信息，读者按照自己的习惯选择使用即可。

如果删除了表 EMP，则视图将变为无效过程，下面我们测试这个过程，首先删除表：

```
SQL> drop table emp;

Table dropped.
```

此时，由于过程 IncreSal 所依赖的表已经不存在，所以过程的状态已经变为无效，如实例 9-11 所示。

实例 9-11　查询过程 IncreSal 的状态。

```
SQL> select object_id,object_type,status,created from user_objects
  2  where object_name='INCRESAL';

 OBJECT_ID OBJECT_TYPE         STATUS  CREATED
----------------- -------------- -------------- --------------
     75130  PROCEDURE          INVALID 12-MAR-12
```

那么如何将该过程恢复为有效呢？显然需要恢复表 EMP，下面测试一下恢复表后，相应过程是否自动恢复为有效，如实例 9-12 所示。

实例 9-12　测试恢复表后，相应过程是否自动恢复为有效。

```
SQL> create table emp as select * from empbk;

Table created.

SQL>  select object_id,object_type,status,created from user_objects where
object_name='INCRESAL';

 OBJECT_ID OBJECT_TYPE         STATUS  CREATED
----------------- -------------- -------------- ----------------
     75130  PROCEDURE          INVALID 12-MAR-12
```

以上代码重新从备份表 empbk 中恢复表 emp，此时过程 IncreSal 依然没有自动恢复为有效的状态。我们通过这个实例可以知道过程不是数据库自动维护的，需要 DBA 参与完成。重新编译该过程，如实例 9-13 所示。

实例 9-13　重新编译过程 IncreSal。

```
SQL> alter procedure IncreSal compile;

Procedure altered.

SQL> select object_id,object_type,status,created from user_objects
where object_name='INCRESAL';
 OBJECT_ID OBJECT_TYPE         STATUS  CREATED
----------------- -------------- ------------- ------------------
     75130  PROCEDURE          VALID   12-MAR-12
```

从输出可以知道，过程 IncreSal 经过重新编译后又变为有效状态，处于 VALID 状态的过程可以使用。

9.4　存储过程的IN和OUT参数

存储过程是存储在数据库服务器端的数据库对象，用于完成在数据库服务器端的计算，显然这个计算需要与用户交互才能完成更具体的任务，可以使用 IN 和 OUT 参数来完成过程使用环境与数据库服务器之间的值传递。代码如下所示。

```
Create or replace procedure p1(id in number, name out varchar2(20))
```

其中 IN 表示输入参数，而 OUT 表示输出参数，在上面的过程定义中，形式参数 ID 为用户 ID，形式参数 NAME 为用户名称，该过程的输入为用户 ID，输出为用户名 NAME。在具体调用该过程时，需要指定实参来代替输入值，通过定义与输出对应的变量来获得输出结果。

下面我们先给出一个实例，这样读者就可以获得更直观的认识，如实例 9-14 所示。

实例 9-14　创建带 IN 和 OUT 参数的过程。

```
SQL> l
  1  create or replace procedure search_name
  2    (id in number,f_name out varchar2,sal out number)
  3   as
  4     begin
  5       select first_name,sal into f_name,sal
  6        from employees
  7        where employee_id=id;
  8   exception
  9     when others
 10       then
 11         dbms_output.put_line('Error!');
 12* end find_name;
SQL> /

Procedure created.
```

在上例中，我们创建了过程 search_name，该过程的输入参数为 ID，经过过程主题的计算返回员工名和相应的工资，并且在定义数据类型时，没有任何约束，数据的约束是在传入值时完成的。下面我们调用该过程，通过一个匿名过程来调用，如实例 9-15 所示。

实例 9-15　通过匿名过程调用过程 search _name。

```
SQL> set serveroutput on;
SQL>declare
  2  first_name varchar2(20);
  3  salary     number(8,2);
  4  begin
  5   search _name(206,first_name,salary);
  6  dbms_output.put_line('id 206' ||first_name||' salary is '||salary);
  7* end;
SQL> /
id 206William salary is

PL/SQL procedure successfully completed.
```

在以上过程中，我们调用了过程 search _name(206,first_name,salary)，注意参数的调用次序，

此时的实参和形参的位置对应，这是最常用的一种形参与实参的匹配方式。如果调用过程时形参位置没有值，则使用该数据类型的默认值。

下面总结一下过程的三种参数类型。

- IN：给过程传值，可以是常量、字面值或者表达式，该参数是只读的。
- OUT：给过程返回值，要求必须是变量，不能使用默认值，该参数是只写的。
- IN OUT：给程序传值，同时是过程返回值，该参数只能是变量。

9.5　存储过程的脚本创建

创建存储过程可以使用 SQL*Plus 工具，也可以在 Windows 的记事本中编辑，当使用记事本编辑存储过程时，需要将它保存为一个后缀为.SQL 的脚本文件，最后使用 SQL*Plus 执行该脚本文件。使用脚本文件的方式修改起来比较方便。

当然，也可以使用 SQL*Plus 工具直接输入创建过程的指令。下面我们使用记事本工具定义一个存储过程。该存储过程的名字为 INSERT_DEPT，作用是向 DEPT 表中插入数据，文件内容如图 9-2 所示。

insert_dept - 记事本

文件(F)　编辑(E)　格式(O)　查看(V)　帮助(H)

```
create or replace procedure insert_dept(no in number,
name varchar2,loc varchar2)

is
   dept_number number(2) :=no;
   dept_name   varchar2(14) :=name;
   dept_loc    varchar2(13) :=loc;
begin
   insert into dept values(dept_number,dept_name,dept_loc);

exception
   when others then
      dbms_output.put_line('errors!');

end;
/
```

图 9-2　在记事本中编辑的存储过程

将其保存为一个.SQL 脚本文件，然后执行该脚本文件，创建过程，此时该存储过程作为数据库对象保存在数据字典中。在当前模式下就可以使用该存储过程。我们保存该文件名为 insert_dept.sql，保存在 F 盘的根目录下。然后执行该脚本文件，代码如下所示。

```
SQL> @f:\insert_dept.sql;

过程已创建。
```

使用数据字典 USER_OBJECTS 查看存储过程 INSERT_DEPT 是否已创建成功，如实例 9-16 所示。

实例 9-16　使用数据字典查看存储过程 INSERT_DEPT 的信息。

```
SQL> col object_name for a20
```

```
SQL> select object_name,object_type,created,status
  2  from user_objects
  3  where object_type = 'PROCEDURE'
  4* and object_name LIKE 'INSERT%'

OBJECT_NAME          OBJECT_TYPE        CREATED          STATUS
----------------- ------------- ------------------------ --------------
INSERT_DEPT           PROCEDURE          20-8 月 -09       VALID
```

从输出可以看出，存储过程 INSERT_DEPT 创建成功，而且 STATUS 为 VALID，所以该存储过程是有效的，当前可以使用。

9.6　存储过程的权限

在创建存储过程时，必须保证当前的用户具有创建存储过程的权限，创建存储过程的权限有两种：一个是 CREATE PROCEDURE，另一个是 CREATE ALL PROCEDURE。前者说明当前用户只能创建自己的存储过程，而后者表示当前用户可以创建任何用户模式下的存储过程。下面我们查看一下 SCOTT 用户的系统权限，如实例 9-17 所示。

实例 9-17　查看 SCOTT 用户的系统权限。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> select *
  2  from user_sys_privs;

USERNAME              PRIVILEGE                    ADM
----------------- ---------------------------------------------- ----------
SCOTT                     UNLIMITED TABLESPACE      NO
SCOTT                     CREATE TRIGGER            NO
SCOTT                     CREATE ANY TRIGGER        NO
```

可见当前用户 SCOTT 没有创建存储过程的权限，所以需要 DBA 授权给 SCOTT 用户，如实例 9-18 所示。

实例 9-18　向 SCOTT 用户授予创建存储过程的权限。

```
SQL> conn system/oracle@orcl
已连接。
SQL> grant create procedure to scott;

授权成功。
```

输出显示授权成功，此时再次使用数据字典 USER_SYS_PRIVS 查看用户 SCOTT 的系统权限，如实例 9-19 所示。

实例 9-19　查看 SCOTT 用户是否具备创建存储过程的权限。

```
SQL> conn scott/oracle@orcl
已连接。
SQL> col username for a15
SQL> col privilege for a30
SQL> run
```

```
  1  select *
  2* from user_sys_privs

USERNAME        PRIVILEGE                        ADM
-------- ------------------------------------------------ -------------
SCOTT           CREATE PROCEDURE                 NO
SCOTT           UNLIMITED TABLESPACE             NO
SCOTT           CREATE TRIGGER                   NO
SCOTT           CREATE ANY TRIGGER               NO
```

从输出的第一行可以看出用户 SCOTT 具有 CREATE PROCEDURE 的权限。当然 DBA 也可以授予 SCOTT 用户创建任何模式下的存储过程的权限，如实例 9-20 所示。

实例 9-20　授予用户 SCOTT 创建任何模式下的存储过程的权限。

```
SQL> grant create any procedure to scott;

授权成功。
```

读者可以通过数据字典 USER_SYS_PRIVS 查看是否成功授权。

9.7　本章小结

本章介绍了 PL/SQL 编程中十分重要的一个数据库对象——存储过程，在实际项目中，使用存储过程的现象十分普遍，因为它可以简化客户端的编程，将复杂的程序逻辑计算迁移到数据库服务器端，并且过程一旦调用就可以被反复使用。本章首先引入了存储过程的概念，然后通过一个实例体验存储过程，对于复杂的存储过程，为了提高交互性往往需要使用输入输出参数，在创建存储过程时不但可以直接在 SQL*Plus 环境下编写，也可以通过预先编写的脚本执行。

第 10 章

函　数

对于学习过任何一门高级语言的读者来说，函数这个概念应该十分熟悉，尤其是在面向过程的语言中，都是使用函数来处理问题的。在面向对象的高级语言中则把函数封装在类中，PL/SQL 中的函数同任何高级语言的函数功能类似，用于完成特定的需求任务。在 PL/SQL 中读者或许已经熟悉了存储过程这个数据库对象，二者存在相似之处，但是二者的最大区别是：函数可以返回数据类型，而存储过程不需要。下面将介绍 PL/SQL 中的函数概念、语法规则以及如何创建和使用函数。

10.1　什么是函数

函数是执行某种功能的代码实体，用户调用函数后输入适当的参数或者不输入参数，函数将执行计算过程，输出计算结果，函数的功能模拟如图 10-1 所示。



图 10-1　函数功能示意图

函数执行某种计算，这种计算方法由函数的功能决定。Oracle 提供了丰富的函数用于扩展数据库的功能，如字符处理函数 LOWER()、数学运算函数 COUNT(*)、ACOS(n)计算参数的反余弦角度值等。

虽然，Oracle 定义了大量的函数，可方便用户管理和维护数据库，但是毕竟现实的需求是多种多样的，所以 Oracle 才允许用户自定义函数。

10.2　创建自定义函数

本节以自定义函数 area 为例进行讲解，该函数的作用是计算给定半径的圆的面积，用户调用该函数时输入半径参数，即可得到计算结果。创建函数的过程如实例 10-1 所示。

实例 10-1　创建自定义函数 area。

```
SQL> CREATE OR REPLACE FUNCTION area(f float)
  2  RETURN float
```

```
  3  IS
  4  BEGIN
  5  RETURN 3.14*(f*f);
  6  END area;
  7  /

函数已创建。
```

因为函数是一种 PL/SQL 程序单元，所以其定义满足 PL/SQL 代码块结构的定义，首先声明函数 FUNCTION，即 CREATE OR REPLACE FUNCTION，随后是函数名以及函数参数。IS 后可以有变量声明，但不是必须的，随后是执行部分，最后的‘/’表示编译该函数。

一旦函数创建成功，就可以使用该函数，为了确保已经成功创建函数 area，可以使用数据字典 USER_OBJECTS 进行查询，如实例 10-2 所示。

实例 10-2　查询当前用户所拥有的函数。

```
SQL> col object_name for a20
SQL> select object_name,object_type,created,status
  2  from user_objects
  3* where object_type = 'FUNCTION'

OBJECT_NAME          OBJECT_TYPE         CREATED          STATUS
---------------- -------------- -------------------------------- -------
FTOCORACLE           FUNCTION            15-8 月 -09       VALID
AREA                  FUNCTION            18-8 月 -09       VALID
```

可见函数 area 创建成功，且状态为 VALID，所以下面就使用该函数计算给定半径的圆的面积，如实例 10-3 所示。

实例 10-3　调用函数 area。

```
SQL> select area(4) from dual;

   AREA(4)
-------------------
     50.24
```

创建自定义函数的语法如下所示。

```
CREATE [OR REPLACE] FUNCTION function_name
 ([argument [{IN|OUT}] datatype ][,…])
 RETURN datatype {IS|AS}
function_body
```

首先是指定创建或替换函数，接着是函数名和参数，参数可以是输入给该函数的（IN），也可以是输出给其他对象的（OUT），参数的个数没有限制，当然也可以没有参数，但是函数必须有返回值，需要在函数定义中明确地指定返回的数据类型 RETURN datatype。

10.3　创建作用于表的函数

在学习了自定义函数的语法规则后，下面创建一个操作表对象的函数。该函数的作用是通过一个员工号，读取员工的工资和姓名，如果该员工的工资大于 4000，则输出该员工的工资数和姓

名，然后返回一个变长字符变量，如实例 10-4 所示。

实例 10-4　创建操作表对象的函数 findwealthier。

```
SQL> create or replace function findwealthier(empnumber number)
  2  return varchar2 is
  3     salary number;
  4     employeename varchar2(20);
  5   begin
  6     select ename,sal into employeename,salary
  7     from emp
  8     where empno = empnumber;
  9    if salary>=4000 then
 10       return 'make good salary';
 11       dbms_output.put_line('salary is'
 12                           ||salary
 13                           ||'name is '
 14                           ||employeename);
 15     else
 16       return 'bad';
 17       dbms_output.put_line('salary is'
 18                           ||salary
 19                           ||'name is '
 20                           ||employeename);
 21    end if;
 22  end findwealthier;
 23  /

函数已创建。
```

同样，我们验证一下是否成功创建该函数，如实例 10-5 所示。

实例 10-5　查询函数 findwealthier 是否创建成功。

```
SQL> select object_name,object_type,created
  2  from user_objects
  3  where object_type = 'FUNCTION'
  4  and object_name like 'FIND%';

OBJECT_NAME          OBJECT_TYPE          CREATED
---------------- ---------------------------------- --------------
FINDWEALTHIER        FUNCTION             18-8 月 -09
```

从上例输出可以看出函数 findwealthier 成功创建，下面可以使用该函数了，所以执行函数 findwealthier，如实例 10-6 所示。

实例 10-6　执行作用于表的函数 findwealthier。

```
SQL> select findwealthier(7839) from dual;

FINDWEALTHIER(7839)
-----------------------------------------------
make good salary

salary is5000name is KING
```

在函数参数中，我们输入了参数 7839，该参数表示一个员工号，而查询结果表明，该员工是

高收入员工，该员工工资为 5000，姓名为 KING。显然通过调用函数可方便用户的操作，用户可以灵活地使用自定义函数根据业务需求定制自己所需的功能函数。

10.4 创建自定义的Java函数

在 Oracle 中支持使用 Java 语言编写的函数。对于习惯于使用 Java 语言的人，或者在已经通过 Java 语言编写了功能强大的类的环境下，可以采用下面这种方式定义 Java 函数。

1．创建包含要发布的公共静态方法的 Java 类

此时，需要创建一个类，该类中包含要发布的 Java 函数，并且要求该方法是公共（Public）的和静态（Static）的，定义如下 Java 类。

```
public class AreaJava{
  public static float areaj(float f){
     float area = 3.14*(f*f);
     return area;
  }
}
```

在该类中包含一个公共的静态方法 area，保存该类为 AreaJava.java 文件，然后编译该类，代码如下所示。

```
F:\>javac AreaJava.java

F:\>
```

此时在 F 盘的根目录下创建了名为 AreaJava.java 的.class 文件，接下来加载该文件到 Oracle。

2．将编译后的.class 文件加载到 Oracle

此时将该类加载到 SCOTT 用户模式，之前必须授予用户 SCOTT JAVAUSERPRIV 特权。授权过程如实例 10-7 所示。

实例 10-7 授予用户 SCOTT 的 JAVAUSERPRIV 特权。

```
F:\>sqlplus /nolog

SQL*Plus: Release 17.2.0.1.0 - Production on 星期二 8 月 18 21:07:02 2009

Copyright (c) 1082, 2005, Oracle.  All rights reserved.

SQL> conn system/oracle@orcl
已连接。
SQL> grant javauserpriv to scott;

授权成功。
```

然后把类 AreaJava.class 加载到 SCOTT 用户模式，如实例 10-8 所示。

实例 10-8 加载 Java 类文件到 Oracle 的 SCOTT 模式。

```
F:\>loadjava -verbose -schema scott -thin -user scott/oracle@localhost:1521
: orcl AreaJava.class
```

```
arguments: '-verbose' '-schema' 'scott' '-thin' '-user' 'scott/oracle@localhost:
 1521:orcl' 'AreaJava.class'
creating : class SCOTT.AreaJava
loading  : class SCOTT.AreaJava
Classes Loaded: 1
Resources Loaded: 0
Sources Loaded: 0
Published Interfaces: 0
Classes generated: 0
Classes skipped: 0
Synonyms Created: 0
Errors: 0
```

显示加载成功，整个加载过程只加载了一个类，没有错误发生。

3. 创建一个 PL/SQL 封装

为了调用该 Java 函数必须创建一个 PL/SQL 封装，此处建立一个 PL/SQL 函数，该函数封装了 Java 函数，如实例 10-9 所示。

实例 10-9 创建 Java 函数的 PL/SQL 封装。

```
SQL> create or replace function areajava(f float) return float
  2  as language java name 'AreaJava.areaj(double) return double';
  3  /

函数已创建。
```

下面验证该函数是否记录在数据字典中，查询数据字典 USER_OBJECTS，如实例 10-10 所示。

实例 10-10 验证 Java 函数。

```
SQL> col object_name for a20
SQL> select object_name,object_type,created,status
  2  from user_objects
  3  where object_type ='FUNCTION';

OBJECT_NAME          OBJECT_TYPE         CREATED        STATUS
---------------- --------------- ------------------------ ----------------
FTOC                  FUNCTION             15-8 月 -09     VALID
FTOCORACLE            FUNCTION             15-8 月 -09     VALID
AREA                  FUNCTION             18-8 月 -09     VALID
FINDWEALTHIER         FUNCTION             18-8 月 -09     VALID
AREAJAVA              FUNCTION             18-8 月 -09     VALID
MONYEMAKER            FUNCTION             18-8 月 -09     INVALID
MONEYMAKER            FUNCTION             18-8 月 -09     VALID

已选择 7 行。
```

可以看到在 OBJECT_NAME 列存在函数 AREAJAVA，并且该行记录的 STATUS 列值为 VALID，说明该函数是有效的，可供用户使用。下面调用自定义的 Java 函数——areajava，如实例 10-11 所示

实例 10-11 调用 Java 函数 areajava。

```
SQL> select areajava(8) from dual;
```

```
  AREAJAVA(8)
----------------------------
200.96
```

10.5 应用RETURN语句

函数是一种 PL/SQL 的子程序，通常使用函数执行计算并返回数值。函数可以被重复调用，通过输入不同的参数完成不同的计算需求，如实例 10-12 所示。

实例 10-12　创建函数 square。

```
create or replace FUNCTION square (original NUMBER) -- 参数列表
   RETURN NUMBER -- 函数返回数据类型
AS
               -- 声明开始部分
   original_squared NUMBER;
   BEGIN        -- 执行体开始部分
        original_squared := original * original;
   RETURN original_squared; -- 函数返回数据
END;
```

执行该函数：

```
BEGIN
DBMS_OUTPUT.PUT_LINE(square(170)); -- invocation
END;
/
```

RETURN 语句用于立即结束函数的执行，并返回数据。一个函数可以包含多个 RETURN 语句。

1．触发 RETURN 语句的函数

在 PL/SQL 的函数定义中，每个执行路径必然导致 RETURN 语句的出现，而且每个 RETURN 语句必须明确一个表达式，函数将表达式的值传递给函数标识符，然后返回调用函数的程序逻辑，如实例 10-13 所示。

实例 10-13　函数中的 RETURN 语句。

```
DECLARE
x INTEGER;
FUNCTION f (n INTEGER)
RETURN INTEGER
IS
BEGIN
RETURN (n*n);
END;
BEGIN
DBMS_OUTPUT.PUT_LINE (
'f returns ' || f(2) || '. Execution returns here (1).'
);
x := f(2);
DBMS_OUTPUT.PUT_LINE('Execution returns here (2).');
END;
/
Result:
```

```
f returns 4. Execution returns here (1).
Execution returns here (2).
```

2. 不触发 RETURN 语句的函数

在函数中，我们可以通过过程控制语句来控制程序流程，使其在某种条件下不触发 RETURN 语句，如实例 10-14 所示。

实例 10-14　不触发 RETURN 语句的函数。

```
SQL> CREATE OR REPLACE FUNCTION fun1 (n INTEGER)
  2       RETURN INTEGER
  3     IS
  4        BEGIN
  5          IF n = 0 THEN
  6           RETURN n+1;
  7          ELSIF n = 1 THEN
  8           RETURN n*n;
  9          END IF;
 10        END;
 11  /

函数已创建。
```

在上例中，我们创建了一个函数 fun1，该函数的输入参数为整数，返回值也为整数，在函数的可执行部分，通过 IF 语句判断是否返回值，如果满足条件，则返回，下面我们执行该函数，此时满足返回条件，如实例 10-15 所示。

实例 10-15　调用函数。

```
SQL> begin
  2  dbms_output.put_line(fun1(0));
  3* end;
SQL> /
1

PL/SQL 过程已成功完成。
```

因为此时函数的输入值为 0，满足第一个 IF 条件，此时返回 0+1 的计算结果。下面我们执行一个不满足条件的函数调用，将函数的输入值设为 17，没有 IF 条件满足，此时调用该函数会提示出现错误，如实例 10-16 所示。

实例 10-16　调用函数出错示例。

```
SQL> l
  1  begin
  2  dbms_output.put_line(fun1(17));
  3* end;
SQL> /
begin
*
第 1 行出现错误:
ORA-06503: PL/SQL: 函数未返回值
ORA-06512: 在 "SYS.FUN1", line 17
ORA-06512: 在 line 2
```

我们知道，此时的函数执行部分没有返回值，但显然该函数需要一个 INTEGER 类型的返回值，所以系统报错。

3. 包含多个 RETURN 语句的函数

下面创建包含多个 RETURN 语句的函数，如实例 10-17 所示。

实例 10-17　创建包含多个 RETURN 语句的函数。

```
SQL> CREATE OR REPLACE FUNCTION fun2 (n INTEGER)
  2         RETURN INTEGER
  3        IS
  4           BEGIN
  5             IF n = 0 THEN
  6              RETURN n+1;
  7             ELSIF n = 1 THEN
  8              RETURN n*n;
  9             ELSE
 10              RETURN n*n*n;
 11             END IF;
 12          END;
 13  /

函数已创建。
```

此时，我们在函数的执行部分通过 ELSIF 子句控制任何 N 值的输入都将导致 RETURN 语句。下面我们调用该函数，如实例 10-18 所示。

实例 10-18　调用函数 fun2。

```
SQL> begin
  2    for i in 0..5 loop
  3       dbms_output.put_line(fun2(i));
  4    end loop;
  5  end;
  6  /
1
1
8
27
64
125

PL/SQL 过程已成功完成。
```

通过调用该函数，在多个 RETURN 语句后都返回了相应数值并打印。

10.6　创建复杂函数

在 SCOTT 用户下创建函数 if_id_exist，如实例 10-19 所示。

实例 10-19　创建复杂函数。

```
SQL> connect scott/oracle
```

```
已连接。
SQL> create or replace function if_id_exist(emp_id in number)
  2     return boolean
  3     as
  4       var_emp_count number;
  5         begin
  6           select count(*)
  7           into var_emp_count
  8           from employees
  9             where employee_id = emp_id;
 10         return 1 = var_emp_count;
 11     exception
 12         when others then
 13           return false;
 14     end if_id_exist;
 15    /

函数已创建。
```

该函数的功能是测试指定的某个 employee_id 是否存在，如果存在就返回 true，如果不存在或者发生异常，则返回 false。我们通过实例 10-20 查看该函数的参数含义。

实例 10-20　查看函数的参数定义。

```
SQL> desc if_id_exist;
FUNCTION if_id_exist RETURNS BOOLEAN
参数名称                  类型                        输入/输出默认值?
-------- ----------------------------------- ------------------------
 EMP_ID                        NUMBER                     IN
```

在下面的应用程序中，我们会使用这个功能来判断该员工是否存在，从而使得程序在逻辑上沿着预期的目标前进。例如向表 employees 中插入数据时，会发生用户输入的员工 ID 与表中的员工 ID 相同的情况。此时，在插入数据前，可以使用该函数进行判断，如果存在，则提示用户错误，如果不存在，则插入数据。下面我们设计一个存储过程，该过程的输入参数为预插入表中的数据，在该过程中使用函数 if_id_exist 进行判断。为了简化实例，我们在 SCOTT 用户下创建一个表 employees，如实例 10-21 所示。

实例 10-21　创建表 employees。

```
SQL> connect sys/oracle as sysdba
已连接。
SQL> create table scott.employees as select employee_id,salary,first_name, last_name
  2  from hr.employees;

表已创建。

SQL> desc scott.employees;
 名称                                       是否为空?    类型
 ---------------------- --------- ----------------------------
 EMPLOYEE_ID                                         NUMBER(6)
 SALARY                                              NUMBER(8,2)
 FIRST_NAME                                          VARCHAR2(20)
 LAST_NAME                                NOT NULL   VARCHAR2(25)
```

通过上面的操作已在 SCOTT 用户下创建了表 employees，同时查看了该表的结构，下面创建

一个过程，通过调用该过程向表 employees 插入数据，如实例 10-22 所示。

实例 10-22　创建过程 insert_emp。

```
SQL> create or replace procedure insert_emp(
  2  emp_id in number,emp_salary in number,
  3  first_name in varchar2(20),last_name in varchar2(25))
  4  is
  5   if_id boolean;
  6  begin
  7   i_id := if_id_exist(emp_id);
  8  if i_id then
  9    dbms_output.put_line('emp_id exist');
 10  else
 11   insert into employees values(emp_id,emp_salary,first_name,last_name);
 12  end if;
 13  end;
 14  /
```

在该过程中调用了我们创建的复杂函数，通过复杂函数的调用完成程序中复杂的逻辑判断和求值。

10.7　处理函数中的异常

函数是一个完整的功能实体，因此需要通过处理函数执行过程中的异常，使得函数自身更加健壮，下面我们给出实例 10-23。

实例 10-23　创建函数 show_ename。

```
SQL>create or replace function show_ename
2    (v_empno scott.emp.empno%type)
3     return varchar2
4       is
5         v_ename  varchar2(10);
6     begin
7       select ename into v_ename from scott.emp
8        where empno = v_empno;
9        return v_ename;
10  exception
11   when no_data_found
12    then
13      return ('the ename is not in the emp');
14   when others
15    then
16      return ('error in running show_ename');
17end;
SQL>
```

在上例中，我们创建了一个函数，其作用是通过 empno 获得员工的姓名，关键是在函数体后使用了异常处理，从而增加了函数的健壮性和可读性。下面我们通过一个匿名过程调用该函数，代码如下所示。

```
SQL>declare
 2  v_name varchar2(50);
```

```
 3  begin
 4    v_name := show_ename(4899);
  5    dbms_output.put_line(v_name);
 6  end;
SQL>/
the ename is not in the emp
```

上例中，我们通过匿名过程调用了函数 show_ename()，此时函数会返回员工名，如果不存在，则执行函数的 Exception 部分，打印一行信息，然后退出函数，结束匿名过程。在上例中员工 empno 为 4899，而此员工号根本不存在，所以函数体中的异常部分被执行，即打印一行消息。

10.8　本章小结

本章介绍了 PL/SQL 中的函数概念、语法规则以及如何创建和使用函数的方法。PL/SQL 中的函数同任何高级语言的函数功能类似，可以用于完成特定的需求任务，并返回读者需要的数据类型，以便读者进行下一步的数据处理工作。

第 11 章

包

包是 Oracle 提供的一种程序逻辑单元，Oracle 拥有自己内嵌的编译好的包，使用这些包可以实现监控以及备份等工作，极大地方便了用户对于数据库的管理以及用户可以根据自己的业务逻辑需要创建各种逻辑功能包，以供程序员调用，从而将复杂的业务逻辑放在数据库的服务器端完成，而不需要在某种高级语言中编写复杂的逻辑关系和重复的 SQL 代码。

11.1 包的创建

包是 PL/SQL 中多个程序单元的逻辑组合，它将过程组合在一个包内容，以供用户调用。使用包后，不需要程序员频繁地修改应用程序，就可保持程序逻辑的完整性，对包中的过程进行重新定义或者编译，以便修改部分功能，从而更好地实现业务逻辑。使用包的好处如下。

- 在程序设计时，程序员可以通过调用完成某种业务逻辑的包来简化编程。
- 包被加载到 SGA 后，便不再需要重新加载，从而减少了每次调用的加载时间。
- 包可以增强安全性，通过创建私有过程或者函数来实现业务逻辑和数据的隐藏。

包中包含：公有变量、私有变量。

- 公有变量：在包的声明中声明的变量称为公有变量，公有变量在整个会话周期中存活，可以在整个用户会话中被调用。
- 私有变量：私有变量是在包的其他主体，如过程中定义的局部变量，私有变量只对包中的主体（如过程）可用，存活于整个会话周期。

创建包需要遵循相应的规范，包规范是包含包的主体对象说明、变量说明、游标说明，但是不包含过程或函数的可执行代码，只是用于告诉包这个过程或函数在包中需要定义，这是创建包的规范，具体代码如下所示。

```
PACKAGE pkg_name
  Is
     [ 变量或者类型声明 ]
     [ 游标声明 ]
     [ 主体对象声明（如函数、过程等）]
END  pkg_name
```

上面仅仅是创建了包规范，即告诉包中存在什么，但是这些对象具体能干什么，相互之间的关系没有明确的定义代码，而这些需要在包体中实现，下面是创建包体的规范。

```
PACKAGE BODY pkg_name
 Is
 [ 变量或者游标声明 ]
 [游标相关 SELECT 语句声明]
 [主体对象的 body 声明]
 [Begin
   可执行代码 ]
 [ Exception
    异常处理 ]
 END [pkg_name];
```

在创建了包规范并创建包体之后，这个包就可以使用了，具体可执行的部分是调用包的函数或者过程来实现具体功能。调用包中元素的方法如下。

```
Pkg_nam.element_name.
```

这种方式适用于在包的外部调用包内的元素，如果从包内调用内部的元素，则可以直接调用，不必使用包名的形式。

下面我们通过具体的实例来说明如何创建包规范以及如何使用包，该包包含全局变量、游标这两个过程，但是这两个过程的名称相同，如何实现同名过程的重载呢？如实例 11-1 所示。

实例 11-1　创建包。

```
SQL> create or replace
  2  package emp_pkg is
  3  var_empno  number;  --公有变量
  4   procedure add_emp(e_number number,e_name varchar2,e_date date);
  5   procedure del_emp(e_number number);
  6   procedure del_emp(e_name varchar2);
  7   procedure raise_sal(e_number number,salary number);
  8  end;
  9  /
程序包已创建。
```

在该包中存在一个变量 var_empno，该变量在创建包规范时定义，属于公有变量，拥有 4 个过程：add_emp、raise_sal 以及两个同名过程 del_emp，同名过程可通过不同的参数加以区分。在上例中，我们声明了一个包，但是包还没有什么作用，仅仅是存在的一个名字，而具体内容是空的，即包中的过程没有具体可执行的代码。如果希望包发挥作用，需要在过程中具体编写可执行的代码，下面我们开始创建包体，如实例 11-2 所示。

实例 11-2　创建包体。

```
SQL> create or replace
  2  package body emp_pkg is
  3  var_inner_number  number := 0;  --私有变量
  4   procedure add_emp(e_number number,e_name varchar2,e_date date) is
  5    begin
  6    insert into emp (empno,ename,hiredate)values(e_number,e_name, sysdate);
  7    end ;
  8   procedure del_emp(e_number number) is
  9    begin
 10      delete from emp where empno = e_number;
 11    end ;
 12   procedure del_emp(e_name varchar2) is
 13    begin
```

```
 14     delete from emp where ename=e_name;
 15    end ;
 16   procedure raise_sal(e_number number,salary number) is
 17    begin
 18      update emp set sal=salary where empno=e_number;
 19    end ;
 20
 21  begin
 22    select empno into var_empno  from emp
 23    where hiredate = (select min(hiredate) from emp);
 24  end;
 25  /

程序包体已创建。
```

在上例创建包体的过程中给出了过程的具体代码，通过 begin…end 来标识一个过程的可执行代码，并且通过 begin…end 来为公有变量 var_empno 赋值。这个值可以被会话中的其他对象调用，同时在创建包体时声明了一个变量 var_inner_number，在创建包体时创建的变量称为私有变量，私有变量只能在包中调用，而不能在包外使用。

11.2 包的调用及过程重载

包体一旦创建成功，此时包的元素就可以被调用，下面我们调用包 emp_pkg 的过程 add_emp 向表 emp 中新增一个员工，如实例 11-3 所示。

实例 11-3 调用包 emp_pkg 的过程。

```
SQL> exec emp_pkg.add_emp(1001,'name1',sysdate);

PL/SQL 过程已成功完成。
```

在过程调用成功后，可通过查询表 emp 来验证过程 add_emp 的调用结果是否成功，如实例 11-4 所示。

实例 11-4 验证过程 add_emp 的调用结果。

```
SQL> select ename,hiredate from emp where empno=1001;

ENAME      HIREDATE
---------- --------------
name1      10-9 月 -12
```

在包的调用中，我们使用了过程名。必须注意我们在包规范中声明了两个同名过程——del_emp，在调用这个过程时，Oracle 为了区分具体是哪个过程，需要使用参数来进行区分。

例如包 emp_pkg 中包含两个过程：del_emp(e_name varchar2)和 del_emp(e_number number)，Oracle 允许在同一个包中定义名称完全相同的过程，但是为了区别过程，需要定义不同的过程参数，这样在调用包的过程时，编译器就可以根据参数的不同而调用对应的过程。两个过程的定义如下。

1．过程 1

```
  procedure del_emp(e_number number) is
    begin
```

```
        delete from emp where empno = e_number;
       end ;
```

该过程的参数为 e_number，要求根据员工号删除员工记录。

2. 过程 2

```
     procedure del_emp(e_name varchar2) is
      begin
        delete from emp where ename=e_name;
      end ;
```

该过程的参数为 e_name，要求根据员工名称删除员工记录。这两个过程的名称相同，但是二者的参数不同，这种现象称为函数重载。

11.3　包的私有过程与函数

私有过程是指只能在包内部调用的过程，这样既实现了信息的安全，同时也丰富了包内部业务逻辑的实现，为更加灵活的编程提供了方便。通过在包中定义私有函数和过程，实现信息的隐藏。这些过程或者函数只能在包内被调用，称为包的私有对象。下面重新编译包 emp_pkg 的包体。我们在该包中增加两个私有元素：一个是私有函数 emp_count，另一个是私有过程 show_emp_count。下面我们重新编译包 emp_pkg，如实例 11-5 所示。

实例 11-5　重新编译包 emp_pkg。

```
SQL> create or replace
  2   package body emp_pkg is
  3   var_inner_number  number := 0;
  4    procedure add_emp(e_number number,e_name varchar2,e_date date) is
  5     begin
  6    insert into emp(empno,ename,hiredate)values(e_number,e_name, sysdate);
  7     end ;
  8    procedure del_emp(e_number number) is
  9     begin
 10       delete from emp where empno = e_number;
 11     end ;
 12    procedure del_emp(e_name varchar2) is
 13     begin
 14       delete from emp where ename=e_name;
 15     end ;
 16    procedure raise_sal(e_number number,salary number) is
 17     begin
 18       update emp set sal=salary where empno=e_number;
 19     end ;
 20    --下面是私有函数的定义
 21    function emp_count
 22       return number
 23     is
 24     var_count number;
 25    begin
 26      select count(*) into var_count
 27       from emp;
 28      return var_count;
```

```
29    exception
30       when others
31        then
32          return(0);
33     end emp_count;
34     --下面是私有过程的定义，该过程调用私有函数 emp_count
35    procedure show_emp_count
36    is
37     var_count number;
38    begin
39     var_count := emp_count;
40     dbms_output.put_line
41        ('the total number is : '||var_count);
42    end show_emp_count;
43
44    begin
45      select empno into var_empno  from emp
46      where hiredate = (select min(hiredate) from emp);
47    end;
48    /

程序包体已创建。
```

在上例中，我们重新编译了包体 emp_pkg，并且在创建包体时增加了一个函数 emp_count 和一个过程 show_emp_count，函数的作用是计算表 emp 中员工的数量，而过程 show_emp_count 的作用是调用函数 emp_count 的计算结果，并打印输出。因为是私有函数，所以它不能被其他过程调用，否则会报错，如实例 11-6 所示。

实例 11-6　调用包的私有函数 emp_count。

```
SQL> declare
  2  var_count number;
  3  begin
  4  var_count :=emp_pkg.emp_count;
  5  dbms_output.put_line(var_count);
  6  end;
  7  /
var_count :=emp_pkg.emp_count;
                   *
第 4 行出现错误:
ORA-06550: 第 4 行, 第 21 列:
PLS-00302: 必须声明 'EMP_COUNT' 组件
ORA-06550: 第 4 行, 第 1 列:
PL/SQL: Statement ignored
```

下面我们尝试调用包的私有过程 show_emp_count，如实例 11-7 所示。

实例 11-7　调用包的私有过程 show_emp_count。

```
SQL> set serveroutput on;
SQL> exec emp_pkg.show_emp_count;
BEGIN emp_pkg.show_emp_count; END;

              *
第 1 行出现错误:
ORA-06550: 第 1 行, 第 15 列:
PLS-00302: 必须声明 'SHOW_EMP_COUNT' 组件
```

```
ORA-06550: 第 1 行, 第 7 列:
PL/SQL: Statement ignored
```

通过上面的运行实例可以知道，没有在包声明中创建的函数或者过程，而是在包体创建时定义的函数或者过程称为私有的，这些私有对象只能在包内部被调用，而其他任何程序都无法调用，否则会报错。

如果包声明中定义了函数，但是在创建包体时没有定义，同样会报错，包声明修改如下。

```
create or replace
   package emp_pkg is
   var_empno  number;  --公有变量
    procedure test(num number); --增加一个过程
    procedure add_emp(e_number number,e_name varchar2,e_date date);
    procedure del_emp(e_number number);
    procedure del_emp(e_name varchar2);
    procedure raise_sal(e_number number,salary number);
   end;
   /
```

但是在创建包体时，我们并没有定义过程 test，所以当创建包体时，将弹出提示：创建的包体带有编译错误。

然后可通过 show error 指令查看具体的错误信息，如实例 11-8 所示。

实例 11-8　查看具体的错误信息。

```
SQL> show error
PACKAGE BODY EMP_PKG 出现错误:

LINE/COL ERROR
-------- -----------------------------------------------------------------
3/16     PLS-00323: 子程序或游标 'TEST' 已在程序包说明中声明,
         必须在程序包体中对其进行定义。
```

从错误输出可以知道，过程 TEST 由于在包体中已经声明，但是在创建包体时没有定义这个过程，所以出现错误。

包的结构定义如下，从中我们只能看到公有对象，而上面我们定义的私有变量将无法显示，如实例 11-9 所示。

实例 11-9　查看包 emp_pkg 的结构定义。

```
SQL> desc emp_pkg;
PROCEDURE ADD_EMP
参数名称                      类型                    输入/输出默认值?
------ ----------------------------------- -----------------------
 E_NUMBER                      NUMBER                  IN
 E_NAME                        VARCHAR2                IN
 E_DATE                        DATE                    IN
PROCEDURE DEL_EMP
参数名称                      类型                    输入/输出默认值?
------------------------- ----------------- -----------------------
 E_NUMBER                      NUMBER                  IN
PROCEDURE DEL_EMP
参数名称                      类型                    输入/输出默认值?
----------------------------------------- ----------------- ----
 E_NAME                        VARCHAR2                IN
```

```
PROCEDURE RAISE_SAL
参数名称                      类型                    输入/输出默认值?
----------------------------------------- ------------------------
 E_NUMBER                      NUMBER                  IN
 SALARY                        NUMBER                  IN
```

输出只有 4 个过程：ADD_EMP、两个 DEL_EMP 及 RAISE_SAL，而我们定义的私有函数 EMP_COUNT 以及私有过程 SHOW_EMP_COUNT 都没有显示，从而隐藏了包中这些对象的信息，包外的任何数据库对象以及用户都无法调用这些私有对象。

11.4 包的变量和游标

包变量分为私有变量和公有变量，在包规范中创建的变量称为公有变量，这些变量可以被包外的其他对象调用，私有变量是在包体创建时定义的变量，这些变量具有包的私有性，只能在包内存活，即包内的函数或者过程可以调用。在创建包体时可以定义游标，此时的游标是静态的游标，与一个具体的 SELECT 语句关联。

下面我们创建一个包规范，声明公有变量、过程和函数，并使用该包创建变量和游标，如实例 11-10 所示。

实例 11-10　创建包 emp_api。

```
SQL> create or replace package emp_api as
  2   var_date date;
  3   procedure change_sal;
  4   function get_empname
  5    return emp.ename%type;
  6  end emp_api;
  7  /

程序包已创建。
```

上面定义了包规范，它包含一个公有变量 var_date 和一个过程 change_sal，该过程的作用是通过游标获取一个集合，分析符合条件的员工工资，满足条件的增加其工资比例。函数 get_empname 的作用是获得符合条件的员工名称，并打印到屏幕。

下面是创建包体的过程，如实例 11-11 所示。

实例 11-11　创建包体。

```
SQL> create or replace package body emp_api as
  2   procedure change_sal
  3     is
  4        cursor cur_raise_sal
  5         is
  6           select sal,mgr from emp;
  7       begin
  8         for r_raise_sal in cur_raise_sal
  9         loop
 10          update emp
 11           set sal =sal*0.95
 12         where mgr=7698;
 13        -- dbms_output.put_line(cur_raise_sal.sal);
```

```
 14        end loop;
 15     end change_sal;
 16  function get_empname
 17      return emp.ename%type
 18    is
 19      var_empname emp.ename%type;
 20    begin
 21     select ename into var_empname
 22     from emp
 23     where sal= (select max(sal) from emp);
 24     return var_empname;
 25     exception
 26      when others
 27        then
 28      dbms_output.put_line('error.....');
 29  end get_empname;
 30  begin
 31   select sysdate into var_date from dual;
 32  end emp_api;
 33  /

程序包体已创建。
```

在包的过程 change_sal 中，我们编写了一个游标，这个游标与一个具体的 SELECT 语句关联，通过 LOOP 的方式遍历整个游标结果集，搜索符合条件的记录，然后更新表 emp 的相关记录数据。函数 get_empname 的作用是查询表 emp 中工资最高的员工姓名，其中包含一个嵌套子查询。下面测试调用包 emp_api 的函数 get_empname，如实例 11-12 所示。

实例 11-12　调用包 emp_api 的函数。

```
SQL> select emp_api.get_empname from dual;

GET_EMPNAME
--------------------------------------------------
KING
```

执行包 emp_api 的 change_sal 过程，将部门 7698 的所有员工工资减少 5%。我们先查询一下当前部门 7698 的员工工资，如实例 11-13 所示。

实例 11-13　查询当前部门 7698 的员工工资。

```
SQL> select sal,ename  from emp
  2  where mgr=7698;

SAL ENAME
---------- ----------
  1600 ALLEN
  1250 WARD
      1250 MARTIN
      1500 TURNER
       950 JAMES
```

下面执行包的过程。

```
SQL> exec emp_api.change_sal;

PL/SQL 过程已成功完成。
```

验证过程调用结果，如实例 11-14 所示。

实例 11-14　验证过程调用结果。

```
SQL> select sal ,ename from emp
  2  where mgr=7698;

      SAL ENAME
---------- ----------
     1520 ALLEN
     1187.5 WARD
     1187.5 MARTIN
     1425 TURNER
     902.5 JAMES
```

在创建包规范时，为了调用游标，可使用游标变量。游标变量是静态游标的引用，它可以对应不同的 SELECT 语句。游标变量使得程序员将该游标的引用传递给其他的程序单元使用，该游标变量在运行时会动态绑定到对应的 SELECT 语句。

为了创建游标变量，首先需要声明游标变量的对应记录，该记录的数据类型与 SELECT 语句中使用的结果集必须相同。

下面先创建一个包规范，此时我们是在模式 HR 下创建该包规范，操作表为 employees，代码如下所示。

```
  type employees_record_type is record
      (first_name   employees.first_name%type,
      last_name    employees.last_name%type,
       job_id       employees.job_id%type,
       salary       employees.salary%type,
      department_id employees.department_id%type);
```

上面我们定义了记录 employees_record，该记录的结果集的数据类型与表 employees 的对应列数据类型一致。

然后，我们需要声明一个 REF CURSOR 类型，即创建 REF CURSOR 类型的游标变量，下面是声明该类型的示例代码。

```
type employees_cursor is ref cursor     [ return employees_record_type ];
```

上面的粗体字部分是不变的，其中“[ ]”中为可选内容。

下面声明一个游标变量，该变量的返回数据类型为定义的记录 employees_record_type：

```
type employees_cursor is ref cursor return employees_record_type;
```

在包中使用游标变量，如实例 11-15 所示。

实例 11-15　创建包规范。

```
SQL> create or replace  package employees_pkg as
  2   type employees_record_type is record
  3     (first_name   employees.first_name%type,
  4      last_name    employees.last_name%type,
  5      job_id       employees.job_id%type,
  6      salary       employees.salary%type,
  7      department_id employees.department_id%type);
  8
```

```
  9   type employees_cursor is ref cursor return employees_record_type;
 10   procedure get_emp_infor
 11   (emp_id number,
 12    employees_cur in out employees_cursor);
 13   end employees_pkg;
 14  /

程序包已创建。
```

在上例中，我们声明了记录 employees_record_type，同时在包规范部分声明了游标变量 employees_cursor，它的返回类型为记录 employees_record_type。下面我们创建包体，如实例 11-16 所示。

实例 11-16　创建包体。

```
SQL> create or replace  package body employees_pkg as
  2   procedure get_emp_infor
  3   (emp_id number,
  4    employees_cur in out employees_cursor)
  5   is
  6   begin
  7      if emp_id is null
  8       then
  9         open employees_cur for --open for 为不同查询打开相同游标变量
 10       select 'first_name' first_name,
 11              'last_name'  last_name,
 12              null         job_id,
 13              null         salary,
 14              null         department_id from dual;
 15      elsif emp_id is not null then
 16        open employees_cur for --open for 为不同查询打开相同游标变量
 17          select first_name first_name,
 18                 last_name  last_name,
 19                 job_id     job_id,
 20                 salary     salary,
 21                 department_id department_id
 22                from employees
 23               where employee_id=emp_id;
 24    end if;
 25   end get_emp_infor;
 26  end employees_pkg;
 27  /

程序包体已创建。
```

过程 get_emp_infor 的程序逻辑很简单，即从输入的员工 ID 获得相应的员工信息，并使用游标获得这个结果集合，如果为空，则赋予要显示的属性对应的信息（参见程序的第 10~14 行）。

下面详细解释过程 get_emp_infor 如何使用游标变量：在上例的包规范中，声明了一个记录类型 employees_record_type，用于定义游标变量使用的 SELECT 语句获得的结果集合。然后创建了一个游标变量，类型为 REF CURSOR，变量名为 employees_cursor。过程根据输入的参数值返回不同的结果集的游标变量，这些结果集的记录类型都是 employees_record_type。根据 emp_id 的不同值使用 SELECT 语句填充游标变量。

为了验证过程的执行结果，我们先创建一个游标变量类型的 SQL*Plus 变量，代码如下所示。

```
SQL>variable  employess_cr REFCUROSR
```

下面执行过程 get_emp_infor，首先输入的 emp_id 为 100，如实例 11-17 所示。

实例 11-17　执行过程 get_emp_infor。

```
SQL> exec employees_pkg.get_emp_infor(100,:employees_cr);

PL/SQL 过程已成功完成。

SQL> print employees_cr;

FIRST_NAME        LAST_NAME       JOB_ID      SALARY  DEPARTMENT_ID
----------- -------- ------------------ ------------- ---- --

Steven             King              AD_PRES    24000    90
```

下面再传入过程 get_emp_infor 的 emp_id 为 null，用于查看返回结果集存在什么区别，如实例 11-18 所示。

实例 11-18　执行过程 get_emp_infor。

```
SQL> exec employees_pkg.get_emp_infor(null,:employees_cr);

PL/SQL 过程已成功完成。

SQL> print employees_cr;

FIRST_NAME LAST_NAME J S D
---------- --------- - - -
first_name last_name
```

在上例中，我们输入的 emp_id 为空，此时会执行 SELECT 语句填充游标，即使用 SELECT…FROM DUAL 填充游标，而不是从实际的表中获取数据，所以游标结果集合如上例输出所示。

11.5　本章小结

本章我们介绍了包的概念，包是一种逻辑载体，将业务功能相同的元素集中起来，为数据库编程提供了很大方便。使用包可以实现信息隐藏、业务逻辑隐藏等功能。本章通过实例给出了如何创建包规范、如何创建包体，以及如何调用包的过程或者实现函数的具体编程需求，最后介绍了私有元素（过程、函数或者变量），同时介绍了常用的游标变量如何在包中使用。

第 12 章

异 常

在编写程序的过程中，总会有这样或那样的错误发生，这样的程序错误或者发生在编译阶段，或者发生在运行阶段，无论是哪种错误都需要我们及时处理，使得程序按照正确的逻辑顺序执行。在 PL/SQL 中，编写的程序代码同样面临着如何处理这些错误的问题，我们称程序运行过程中的错误为异常。本章我们会介绍用户定义异常和 Oracle 自带的内置异常，并给出相应实例。

12.1 什么是异常

异常是 PL/SQL 语句在运行时或者编译时的错误。为了处理异常，PL/SQL 语句块的某个部分会处理这个异常，在异常处理代码中，程序员可以定义采取什么步骤来处理这个异常。下面我们通过一个典型的实例说明异常，希望读者对异常能有一个直观的体验，然后我们再采取措施处理这个异常，如实例 12-1 所示。

实例 12-1　典型异常。

```
SQL>declare
  2   result number;
  3  begin
  4    result := 1/0;
  5    dbms_output.put_line ('result is '||result);
  6*  end;
SQL> /
declare
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 4
```

在上例中的第 4 行，我们给出分母为 0 的“除法”计算。在该段代码的编译过程中，出现了异常。提示的错误号为 ORA-01476，原因很清楚，即 divisor is equal to zero。对于错误号问题，我们可以通过 oerr 指令查看 Oracle 预定义的异常内容，如实例 12-2 所示。

实例 12-2　通过 oerr 指令查看 Oracle 预定义的异常内容。

```
[oracle@localhost ~]$ oerr ora 01476
01476, 00000, "divisor is equal to zero"
// *Cause:
```

```
    // *Action:
```

下面我们考虑如何解决这个异常。在 Oracle 的 PL/SQL 中，任何 begin…end 块的结尾处都可以处理异常，并且异常会逐层抛出，这个问题将在后面讲解，这里仅介绍如何在执行部分处理该异常，如实例 12-3 所示。

实例 12-3　在执行部分处理异常。

```
SQL> declare
  2   result number;
  3  begin
  4   result := 1/0;
  5   dbms_output.put_line ('result is '||result);
  6   exception
  7    when zero_divide then
  8       dbms_output.put_line('divisor is equal to zero!');
  9*  end;

divisor is equal to zero!

PL/SQL procedure successfully completed.
```

在上述代码中的 begin…end 语句块中，我们给出了异常处理语句，EXCEPTION 标识异常的处理，之后使用 WHEN 子句判断异常类型，这里使用了 Oracle 的内置异常 zero_divide。当我们再次执行该语句时，不会再输出显示错误，执行部分处理了该异常。

在上例中，我们使用了一个语句块 begin…end，在异常发生时，语句自动在本执行语句块内搜索是否存在异常处理代码，恰好我们设计了异常处理代码，使得异常顺利处理。

12.2　异常处理

在单个 begin…end 语句块中，一旦发生异常，程序将在错误处停止，并转到该语句块的末尾搜索是否存在异常处理语句，如果有则处理。对于多层嵌套的语句块中发生的错误，当前语句块的末尾没有异常处理语句，或者有异常处理语句，但是异常类型不匹配，则需要跳出当前的可执行语句块，到外层语句块的末尾继续搜索，直到异常得到处理。需要注意的是，如果整个 PL/SQL 代码都没有异常处理程序，则该异常将交给操作系统环境处理，这显然不是我们需要的，学习异常的目的就是要避免这种情况，可控地管理异常。实例 12-4 演示了异常处理的过程。

实例 12-4　异常处理的过程。

```
SQL> l
  1  declare
  2    result number;
  3  begin              --BLOCK1
  4   dbms_output.put_line ('block 1');
  5    begin            --BLOCK2
  6      dbms_output.put_line ('block 2');
  7       begin         --BLOCK3
  8         result := 1/0;
  9         dbms_output.put_line ('result is '||result);
 10       end;
```

```
 11      exception
 12        when value_error then
 13          dbms_output.put_line(' block 2 :value error!');
 14       end;
 15     exception
 16       when zero_divide then
 17         dbms_output.put_line(' block 1 :divisor is equal to zero!');
 18* end;
```

在上述程序中，我们定义了三个语句块：BLOCK1、BLOCK2 和 BLOCK3，在 BLOCK3 中，我们会触发一个异常，但是在 BLOCK3 的末尾没有异常处理语句。在 BLOCK2 的末尾我们定义了一个异常，但是该异常为 Oracle 预定义的异常 value_error。在 BLOCK1 的末尾我们也定义了一个异常，该异常为 Oracle 预定义的异常 zero_divide。下面我们执行该语句，看看异常在哪个部分得到了处理。

```
SQL> /
block 1
block 2
block 1 :divisor is equal to zero!

PL/SQL procedure successfully completed.;
```

上面是代码的执行结果，显然，输出显示异常在 BLOCK1 中得到处理。下面分析一下这个过程：程序的代码顺序执行，首先输入 BLOCK1 中的第一条可执行语句，打印消息，然后进入第二条语句，而第二条语句就是执行 BLOCK2，然后执行 BLOCK2 的第一条语句，打印消息，进入第二条语句，而第二条语句就是执行 BLOCK3，接着执行 BLOCK3 的第一条语句，此时会触发异常事件。在 BLOCK3 中，发生了异常，此时程序会在 BLOCK3 中搜索异常处理语句，发现没有异常处理语句。然后跳出当前语句块，向次外层语句块 BLOCK2 搜索是否存在异常处理语句，此时没有再次执行 BLOCK2 的第一条可执行语句，而是直接跳转到 BLOCK2 的末尾搜索是否存在匹配的异常处理，发现异常类型不匹配，则继续跳转到外围执行语句块 BLOCK1，注意此时也不会执行 BLOCK3 中的任何可执行语句，而是直接在 BLOCK3 的末尾处搜索是否存在匹配异常，发现与该异常匹配处理语句的执行异常后，此时异常得到处理。

12.3　预定义异常

Oracle 的预定义异常也称为系统异常，这些异常所有的 PL/SQL 语句都可以使用，该类异常在包 STANDARD 中定义。下面我们给出 Oracle 内置的这些预定义异常以及异常的含义，这样读者在使用时就可以查询需要的异常类型，如实例 12-5 所示。

实例 12-5　Oracle 内置的预定义异常。

```
CURSOR_ALREADY_OPEN exception;
    pragma EXCEPTION_INIT(CURSOR_ALREADY_OPEN, '-6511');
  DUP_VAL_ON_INDEX exception;
    pragma EXCEPTION_INIT(DUP_VAL_ON_INDEX, '-0001');
  TIMEOUT_ON_RESOURCE exception;
    pragma EXCEPTION_INIT(TIMEOUT_ON_RESOURCE, '-0051');
  INVALID_CURSOR exception;
```

```
      pragma EXCEPTION_INIT(INVALID_CURSOR, '-1001');
    NOT_LOGGED_ON exception;
      pragma EXCEPTION_INIT(NOT_LOGGED_ON, '-1012');
    LOGIN_DENIED exception;
      pragma EXCEPTION_INIT(LOGIN_DENIED, '-1017');
    NO_DATA_FOUND exception;
      pragma EXCEPTION_INIT(NO_DATA_FOUND, 100);
    ZERO_DIVIDE exception;
      pragma EXCEPTION_INIT(ZERO_DIVIDE, '-1476');
    INVALID_NUMBER exception;
      pragma EXCEPTION_INIT(INVALID_NUMBER, '-1722');
    TOO_MANY_ROWS exception;
      pragma EXCEPTION_INIT(TOO_MANY_ROWS, '-1422');
    STORAGE_ERROR exception;
      pragma EXCEPTION_INIT(STORAGE_ERROR, '-6500');
    PROGRAM_ERROR exception;
      pragma EXCEPTION_INIT(PROGRAM_ERROR, '-6501');
    VALUE_ERROR exception;
      pragma EXCEPTION_INIT(VALUE_ERROR, '-6502');
    ACCESS_INTO_NULL exception;
      pragma EXCEPTION_INIT(ACCESS_INTO_NULL, '-6530');
    COLLECTION_IS_NULL exception;
      pragma EXCEPTION_INIT(COLLECTION_IS_NULL , '-6531');
    SUBSCRIPT_OUTSIDE_LIMIT exception;
      pragma EXCEPTION_INIT(SUBSCRIPT_OUTSIDE_LIMIT,'-6532');
    SUBSCRIPT_BEYOND_COUNT exception;
      pragma EXCEPTION_INIT(SUBSCRIPT_BEYOND_COUNT ,'-6533');
    -- exception for ref cursors
    ROWTYPE_MISMATCH exception;
    pragma EXCEPTION_INIT(ROWTYPE_MISMATCH, '-6504');
    SYS_INVALID_ROWID  EXCEPTION;
    PRAGMA EXCEPTION_INIT(SYS_INVALID_ROWID, '-1410');
    -- The object instance i.e. SELF is null
    SELF_IS_NULL exception;
      pragma EXCEPTION_INIT(SELF_IS_NULL, '-30625');
    CASE_NOT_FOUND exception;
      pragma EXCEPTION_INIT(CASE_NOT_FOUND, '-6592');
    -- Added for USERENV enhancement, bug 1622213.
    USERENV_COMMITSCN_ERROR exception;
      pragma EXCEPTION_INIT(USERENV_COMMITSCN_ERROR, '-1725');
    -- Parallel and pipelined support
    NO_DATA_NEEDED exception;
      pragma EXCEPTION_INIT(NO_DATA_NEEDED, '-6548');
    -- End of 8.2 parallel and pipelined support
```

在上述定义中，我们注意到一个语句“pragma EXCEPTION_INIT(NO_DATA_NEEDED, '-6548');”，此时它调用了一个 Oracle 编译指令 EXCEPTION_INIT，使用它可以定义用户自己的异常错误号和对应的异常，这些内置的异常都是隐式抛出的，只要遇到预定义异常就抛出异常，而不需要再显式地抛出该异常。

每个异常都有一个对应错误号的文本来详细介绍错误内容。对于如下异常：

```
VALUE_ERROR exception;
    pragma EXCEPTION_INIT(VALUE_ERROR, '-6502');
```

可使用 oerr 指令查看错误原因，如实例 12-6 所示。

实例 12-6　使用 oerr 指令。

```
[oracle@localhost ~]$ oerr ora 6502
06502, 00000, "PL/SQL: numeric or value error%s"
// *Cause: An arithmetic, numeric, string, conversion, or constraint error
//         occurred. For example, this error occurs if an attempt is made to
//         assign the value NULL to a variable declared NOT NULL, or if an
//         attempt is made to assign an integer larger than 99 to a variable
//         declared NUMBER(2).
// *Action: Change the data, how it is manipulated, or how it is declared so
//          that values do not violate constraints.
```

12.4　自定义异常

除了 Oracle 的预定义异常外，我们也可以自定义异常，显然自定义异常明显增加了程序员编写程序的灵活性，并且丰富了 Oracle 的异常类型，其实这种开放模式在面向对象语言中都支持，通过用户自定义异常来满足不同用户的实际业务需求。

若要自定义异常，首先要在 DECLARE 声明部分声明异常，并且在异常处理语句中使用 WHEN 子句来捕获对应的异常，这些异常必须在执行语句中可能发生异常的部分显式地抛出异常，如实例 12-7 所示。

实例 12-7　自定义异常语法。

```
DECLARE
 exception_name EXCEPTION;
BEGIN
  ...
  IF 条件 then
    RAISE exception_name;
  END IF;
 EXCEPTION
    WHEN exception_name then
      处理异常
 END;
```

既然是用户定义的异常，显然用户知道在哪里、在什么情况下需要抛出异常，并且是抛出什么类型的异常。在这种程序结构中，往往使用 IF 语句来判断是否需要抛出异常。下面的实例 12-8，有助于读者理解自定义异常是如何定义、触发以及处理的。

实例 12-8　用户自定义异常的触发以及处理。

```
SQL> declare
  2     test_number number;
  3     test_exception exception;
  4   begin
  5       dbms_output.put_line('continue...!');
  6   begin
  7       test_number :='&number';
  8      if test_number >100 then
  9         raise test_exception;
 10      end if;
 11
```

```
 12   exception
 13     when no_data_found then
 14        dbms_output.put_line('no date found!');
 15     when value_error   then
 16        dbms_output.put_line('value error!');
 17     when test_exception then
 18        dbms_output.put_line('test_exception');
 19        test_number := 100;
 20   end;
 21* end;
```

在上例中，我们定义了异常 test_exception，这是用户自定义的异常，并定义了一个 NUMBER 类型的变量 test，在程序的执行过程中需要为该变量赋值，如果该值大于 100，则报错，类似的实例很多，如学生的考试成绩（百分制）不允许输入超过 100 的数值。一旦违反该规则，就抛出用户自定义的异常，然后该异常在当前语句块的末尾处理。下面执行上述代码。

```
SQL> /
Enter value for number: 1000
old   7:     test_number :='&number';
new   7:     test_number :='1000';
continue...!
test_exception

PL/SQL procedure successfully completed.
```

此时，为 test 变量赋予的值为 1000，显然这个值大于 100，违反了先前在 IF 语句中定义的规则，此时会抛出用户自定义异常 test_exception，然后该异常在 EXCEPTION 部分被捕获，并且找到了匹配的异常类型，所以可以成功处理该异常，然后程序继续并且顺序执行外层语句。

如果输入的变量值小于 100，则不会触发异常，此时程序顺序执行外围语句块中的语句，代码如下所示。

```
SQL> /
Enter value for number: 12
old   7:     test_number :='&number';
new   7:     test_number :='12';
continue...!

PL/SQL procedure successfully completed.
```

无论是内置异常还是用户自定义异常，该异常的作用范围仅限于当前的语句块，如在内部语句块中声明的异常必须在内部语句块中抛出，而外部语句块抛出内部语句块中的异常会报错，需要在外部语句块中定义该异常，其实这与数据类型的定义相似，内部语句块定义的数据类型只对内部语句块有效，或者说可见，如实例 12-9 所示。

实例 12-9　异常的有效范围。

```
SQL> declare
  2     test_number number;
  3   begin
  4       dbms_output.put_line('continue...!');
  5     declare
  6         test_exception exception;   --内部语句块声明异常
  7     begin
  8      test_number :='&number';
```

```
  9      exception                  --内部语句块的异常处理
 10        when test_exception then
 11           dbms_output.put_line('test_exception');
 12      end;
 13      if test_number >100 then  --在外部语句块抛出内部语句块声明的异常
 14        raise test_exception;
 15      end if;
 16* end;
```

在上例中，存在两个语句块，我们分别定义为内部语句块和外部语句块，在外部语句块中声明了变量 test，在内部语句块中定义了用户自定义异常 test_exception。在内部语句块中为变量 test 赋值，但是测试该值是否满足规则的语句在外部语句块中执行，并抛出在内部语句块中定义的用户定义类型。显然，我们在外部语句块中抛出了内部语句块定义的异常类型，这是不允许的，外部语句块无法看到这个内部语句块的异常定义，会抛出错误。执行该语句块的结果如下。

```
SQL> /
Enter value for number: 1000
old   8:     test_number :='&number';
new   8:     test_number :='1000';
     raise test_exception;
           *
ERROR at line 14:
ORA-06550: line 14, column 13:
PLS-00201: identifier 'TEST_EXCEPTION' must be declared
ORA-06550: line 14, column 7:
PL/SQL: Statement ignored
```

显然，错误提示 identifier 'TEST_EXCEPTION' must be declared，需要定义该标识符。但是，我们在外部语句块中定义的变量 test 在内部语句块中是可见的。

12.5　异常传播

在异常发生时，程序逻辑会在异常发生处停止，进而跳转到异常处理语句块部分进行处理。但是异常不仅仅发生在可执行部分，也有可能发生在变量声明部分或者异常处理部分，那么此时的异常又是怎么传播的呢？本节我们就来讨论这个问题。

12.5.1　可执行部分发生异常

对于在可执行部分发生的异常，程序会首先通过可执行语句块内部的异常进行处理，如果找不到匹配的异常，再跳转到外部语句块，在外围语句块的异常处理部分继续处理，如果可以处理，则处理，否则继续向外传播，这个过程一直进行，直到将异常传播到主机系统，此时会提示异常没有被捕获。但是作为编程人员是不应该将错误传播到主机系统的，至少要使用 OTHERS 异常来捕获所有不能确定的运行时异常，使得程序逻辑可以继续执行。我们在前面介绍的异常都是在可执行部分发生的异常，这里不再给出实例。

12.5.2 声明部分发生异常

当异常发生在声明部分时，会发生运行时错误。按照我们的理解该异常应该在执行语句的异常处理部分捕获并处理。先看一下实例 12-10，然后分析这样的异常到底传播到了哪里。

实例 12-10 分析异常传播。

```
SQL> declare
  2   var_name varchar2(5)  := 'Abroham';
  3   begin
  4     dbms_output.put_line('var_name : '||var_name);
  5     exception
  6       when others then
  7         dbms_output.put_line(sqlcode);
  8         dbms_output.put_line(sqlerrm);
  9         dbms_output.put_line('inner :error happened...!');
 10*   end;
```

在上例中的声明部分定义了变量 var_name，但是由于字符串缓冲比所赋予的数值要小，所以会发生异常，这是我们故意设计的行为，希望验证 Oracle 在此时是如何处理异常的。在执行语句的尾部我们使用了 EXCEPTION 语句来捕获异常并处理。下面执行该过程，查看异常是如何传播的。

```
SQL> /
 declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 2
```

从过程执行结果来看，异常确实发生了，但是显然我们的 OTHERS 异常没有捕获在过程声明部分发生的运行时错误。对于这个过程，显然异常发生时过程已终止，而异常最终传播到了操作系统环境。

上面的实例只有一个过程，也就是说只有一个 begin…end 语句，下面分析一下如果具有多层 begin…end 可执行部分，内部语句的错误是否会传播到外部语句块，也就是说内部的异常是否会传播到外部语句块处理，如实例 12-11 所示。

实例 12-11 多层 begin…end 可执行部分的异常传播。

```
SQL> ldeclare
  2  begin
  3   dbms_output.put_line('outer...');
  4  declare
  5   var_name varchar2(5)  := 'Abroham';
  6   begin
  7     dbms_output.put_line('var_name : '||var_name);
  8     exception
  9       when others then
 10         dbms_output.put_line(sqlcode);
 11         dbms_output.put_line(sqlerrm);
 12         dbms_output.put_line('inner :error happened...!');
 13   end;
 14  exception
```

```
 15    when others then
 16      dbms_output.put_line(sqlcode);
 17      dbms_output.put_line(sqlerrm);
 18      dbms_output.put_line('outer:error happened...!');
 19* end;
```

在上例中多层的 begin…end 可执行语句块，在内部语句块的声明部分会抛出异常，在内部语句块和外部语句块中都定义了异常处理语句。下面我们执行该过程，查看内部语句在声明部分的异常传播到了哪里。

执行语句块测试异常传播的结果如下：

```
SQL> /
outer...
-6502
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
outer:error happened...!

PL/SQL procedure successfully completed.
```

从输出可以知道，该异常在外部语句块的异常处理部分得以处理，在 OTHERS 异常部分使用了 SQLCODE 和 SQLERRM 打印异常的具体信息。

通过上面两个实例的演示可以清楚地知道，对于发生在声明部分的异常，如果没有外部可执行语句，则传播到主机环境，如果有外部语句且外部语句有相应的异常处理语句，则在外部语句块的异常处理部分处理该异常，否则继续传播到主机环境。发生声明部分异常时的传播流程如图 12-1 所示。

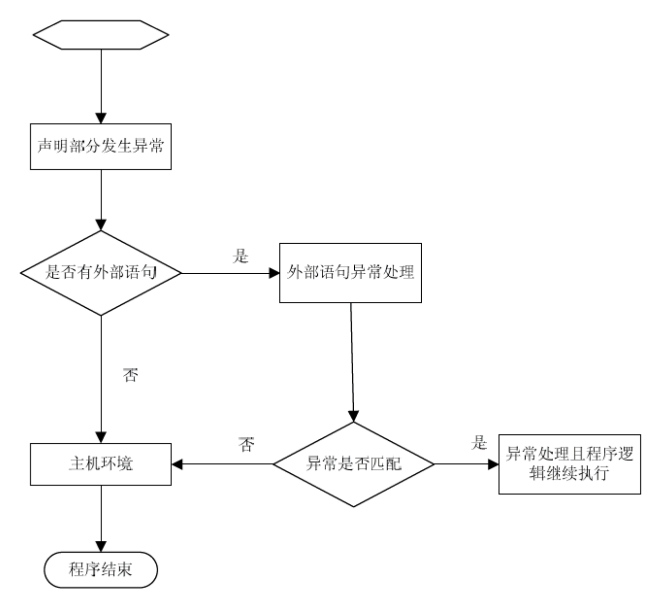


图 12-1　声明部分异常传播流程

上图分析了在声明部分发生异常时异常的传播流程，这里只说明了具有两层 being…end 执行语句块的过程中的异常处理，具有更多层的处理逻辑类似。

通过分析我们可以确定，当内部语句块发生声明部分运行时错误时，该异常会立即传播到外部语句块。如果外部语句块不能处理该异常，则异常会继续向外传播，如果外部语句块也不能处理

该异常，则继续传播到主机环境。

内部异常由运行系统隐式地抛出，如果用户定义的异常使用 EXCEPTION_INIT 过程将 Oracle 错误号和错误消息关联起来，该异常也会隐式地抛出。而对于其他异常则必须显式地抛出，此时就需要使用 Raised 关键字，或者使用 RAISE_APPLICATION_ERROR 过程。Raised 用户定义的异常如实例 12-12 所示。

实例 12-12　Raised 用户定义异常。

```
SQL>declare
  2    my_exp1  exception;
  3    var_name varchar2(10) := '&titile';
  4  begin
  5     if var_name = 'manager' then
  6       raise my_exp1;
  7     end if;
  8    exception
  9     when my_exp1 then
 10      dbms_output.put_line('can not modify manager!');
 11* end;
```

在上例中，我们使用 Raise 抛出了用户自定义的异常来传播该异常，一旦显式抛出该异常后，将在 EXCEPTION 部分执行异常处理。下面我们执行该过程，查看用户定义异常 my_exp1 是否被捕获。

```
SQL> /
Enter value for titile: manager
old   3:   var_name varchar2(10) := '&titile';
new   3:   var_name varchar2(10) := 'manager';
can not modify manager!

PL/SQL procedure successfully completed.
```

执行该过程，提示输入 title，我们输入 manager，此时 IF 判断语句会抛出异常，即使用显式的 Raised 抛出 my_exp1 异常，在程序的尾部的 EXCEPTION 部分捕获异常并处理，我们在此时打印一行消息 can not modify manage，在实际的编程中读者可以根据需要编写具有实际意义的异常处理语句。除了用户自定义异常外，如果读者对于 Oracle 的预定义异常比较熟悉，也可以根据实际需要显式地抛出预定义异常。

下面我们讨论如何再次抛出异常，在内部语句块中处理了异常，然而从内部语句块的异常处理部分可以再次抛出异常，此时可使用 Raise 语句，而不需要指明异常名称，此时的 Raise 含义是将当前异常处理部分处理的异常再次抛出，从外部语句块中接收处理，如实例 12-13 所示。

实例 12-13　异常的再次抛出。

```
SQL> set serveroutput on;
SQL> DECLARE
  2     number_high EXCEPTION;
  3     current_number NUMBER := '&curr_number';
  4     max_number NUMBER :=100;
  5     err_number NUMBER;
  6     BEGIN             --外部语句块
  7       BEGIN           --内部语句块
  8         IF current_number > max_number THEN
```

```
 9         RAISE number_high; -- raise the exception
10        END IF;
11        EXCEPTION
12         WHEN number_high THEN
13                         -- 第一次处理异常 number_high
14      DBMS_OUTPUT.PUT_LINE('number'||current_number||'is out of range.');
15          DBMS_OUTPUT.PUT_LINE ('Maximum number is ' || max_number || '.');
16          RAISE;        --再次抛出异常
17      END;              -- 结束内部语句块
18     EXCEPTION
19       WHEN number_high THEN
20                          -- 再次处理异常
21         err_number := current_number;
22         current_number := max_number;
23       DBMS_OUTPUT.PUT_LINE (
24          'Revising number from'||err_number||'to'||current_number||'.'
25          );
26*    END;
```

在上例中，我们定义了一个用户自定义异常 number_high 以及三个 NUMBER 类型的变量，它们分别是 current_number、max_number、err_number，并且变量 current_number 需要在执行该过程时使用替代变量来赋值。整个过程的执行语句包括两个语句块：外部语句块和内部语句块，在内部语句块中使用 IF 判断 current_number 是否大于 max_number。如果大于则抛出用户自定义异常 number_high，处理异常的语句为提示用户当前的数字超出范围，并打印允许的最大数字，然后使用 RAISE 语句再次抛出异常，此时抛出的异常名为 number_high。在外围语句块继续处理。下面我们执行该过程，分析执行结果。

```
SQL> /
Enter value for curr_number: 101
old   3:    current_number NUMBER := '&curr_number';
new   3:    current_number NUMBER := '101';
number 101 is out of range.
Maximum number is 100.
Revising number from 101 to 100.

PL/SQL procedure successfully completed.
```

在上例中，我们执行了过程，并且输入了数字 101，将该数值赋予 current_number 变量，此时，通过分析程序逻辑可以知道由于 101 大于 100，不满足内部语句块的 IF 语句判断条件会抛出异常，该异常在内部语句块处理后会继续通过 RAISE 语句向外传播，在外部语句块继续处理，上例中的输出 Revising number from 101 to 100 是外部语句块处理的结果。通过执行过程并观察执行结果，可以知道通过内部异常处理语句块的 RAISE 语句，再次抛出了异常 number_high，该异常在外部语句块继续处理。

12.5.3　异常处理部分发生异常

异常是未知的事件。在 PL/SQL 编程中会遇到各种各样的异常，在异常处理部分依然会存在异常发生的可能性，那么对于一个语句块异常部分的异常应该如何处理呢？是内部语句块自己处理，或传播到主机环境还是传播到外部语句块呢？下面我们通过实例 12-14 来验证异常处理部分发生异常的传播路径。

实例 12-14　验证异常处理部分发生异常的传播路径。

```
SQL>declare
  2   var_name varchar2(5)  := 'Mark';
  3   var_number number;
  4   begin
  5      var_number := 1/0;
  6      dbms_output.put_line('var_name : '||var_name);
  7      exception
  8        when others then
  9          var_name := 'Abraham';
 10          dbms_output.put_line(sqlcode);
 11          dbms_output.put_line(sqlerrm);
 12          dbms_output.put_line('inner :error happened...!');
 13*   end;
```

在上例中，语句的可执行部分会触发一个除 0 错误，该错误会传播到可执行语句尾部的 EXCEPTION 异常处理部分。但是在异常处理语句的第一行的可执行语句就发生了异常，那么这个异常应该如何处理呢？下面执行这个过程，用于验证异常是如何处理的。

```
SQL> /
 declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 9
ORA-01476: divisor is equal to zero
```

显然，从错误输出的提示信息可以知道，定义的 OTHERS 异常并没有处理在自身异常部分发生的异常，而是传播到了主机环境。显然这是一个没有外部语句块的过程。如果具有外部语句块，那么在异常处理部分发生的异常是否会传播到外部语句块中处理呢？下面我们给出实例 12-15，通过实例会更直观地看到这类异常的传播途径。

实例 12-15　测试异常的传播途径。

```
SQL> begin
  2   declare
  3   var_name varchar2(5)  := 'Mark';
  4   var_number number;
  5   begin
  6      var_number := 1/0;
  7      dbms_output.put_line('var_name : '||var_name);
  8      exception
  9        when others then
 10          var_name := 'Abraham';
 11          dbms_output.put_line(sqlcode);
 12          dbms_output.put_line(sqlerrm);
 13          dbms_output.put_line('inner :error happened...!');
 14   end;
 15        exception
 16        when others then
 17          dbms_output.put_line('sqlcode : '||sqlcode);
 18          dbms_output.put_line('sqlerrm***: '||sqlerrm||'***sqlerrm');
 19          dbms_output.put_line('outter :error happened...!');
 20* end;
```

在上例中，我们给出了一个外部语句块，内部语句块的代码没有改变，但是在外部语句块中使用了 OTHERS 异常来捕获所有异常，并打印异常的 SQLCODE 和 SQLERRM 信息。通过这些信息可以确认内部语句块在异常处理部分的异常是否得到了处理。下面通过执行该实例进行验证。

```
SQL> /
sqlcode : -6502
sqlerrm***: ORA-06502: PL/SQL: numeric or value error: character string buffer
too small
ORA-01476: divisor is equal to zero***sqlerrm
outter :error happened...!

PL/SQL procedure successfully completed.
```

首先，通过输出信息可以知道，内部语句块在异常处理部分发生的异常在外部语句块被捕获并处理，也就是说内部语句块在异常处理部分的异常被立即传播到了外部语句块去处理。因为外部语句块的异常处理部分可以处理该异常，所以不会继续传播到主机系统。



此时的 SQLERRM 包含两个异常信息：一个是内部语句块的除 0 错误，另一个是除 0 错误发生时，在异常处理部分的字符串缓冲区过小的异常。这两个异常被打包在一起呈现给用户。

12.6 应用RAISE_APPLICATION_ERROR

RAISE_APPLICATION_ERROR 是一个特殊的过程，通过该过程可以为特殊的异常定义具体内容，它属于用户定义异常的范畴。用户可以为自己的应用程序中可能发生的异常定义具有明显实际意义的错误消息。

其语法是“RAISE_APPLICATION_ERROR（错误号，错误消息）”。这里的错误号范围为-20999~-20000。错误消息最多包含 2048 个字符。下面通过实例 12-16 演示其用法。

实例 12-16 RAISE_APPLICATION_ERROR 的用法。

```
SQL> set serveroutput on;
SQL>declare
  2      mgr_id   number(6)  := '&id';
  3      count_num number ;
  4      my_exception exception;
  5  begin
  6      if mgr_id <0 then
  7        raise my_exception;
  8      else
  9         select count(*)
 10            into count_num from employees
 11            where manager_id=mgr_id;
 12           dbms_output.put_line('sum of '||mgr_id||' is '||count_num);
 13     end if;
 14    exception
 15     when my_exception then
 16   dbms_output.put_line('error happened...negative mgr_id is not allowed here!');
 17  end;
 18
```

```
 19*
```

在上例中，我们的目的是计算在表 employees 中指定的 manager_id 的员工数，并计算一个总值，然后打印出来。但是已经估计到用户输入时会发生输入负数的情况，所以定义了一个异常 my_exception，用于在发生输入负数错误时抛出该异常，下面我们执行该过程。先输入值 122 测试结果如下。

```
SQL> /
输入 id 的值:  122
原值    2:     mgr_id   number(6)  := '&id';
新值    2:     mgr_id   number(6)  := '122';
sum of 122 is 8

PL/SQL 过程已成功完成。
```

显然，此时 manager_id 为 122 的记录存在，共有 8 条记录。下面我们输入一个负值，查看一下用户定义异常是否被触发，代码如下所示。

```
SQL> /
输入 id 的值:  -122
原值    2:     mgr_id   number(6)  := '&id';
新值    2:     mgr_id   number(6)  := '-122';
error happened... negative mgr_id is not allowed here!

PL/SQL 过程已成功完成。
```

此时，由于输入的是负值，此时将抛出用户自定义异常，打印异常提示语句。

显然，在输入负数后，程序逻辑判断这个错误抛出异常，在执行语句的结尾处捕获该异常，并且打印异常信息。在上例中，虽然也成功处理了预测的异常，但是显然使用了用户定义的异常对象，然后再异常捕获阶段定义如何处理该异常。现在选择 RAISE_ APPLICATION_ERROR 来完成错误处理，修改后的代码如实例 12-17 所示。

实例 12-17　选择 RAISE_APPLICATION_ERROR 来完成错误处理。

```
SQL>declare
  2      mgr_id   number(6)  := '&id';
  3      count_num number ;
  4      my_exception exception;
  5   begin
  6      if mgr_id <0 then
  7        raise_application_error
  8        (-20001,' error happened...negative mgr_id is not allowed here')
  9      else
 10         select count(*)
 11            into count_num from employees
 12            where manager_id=mgr_id;
 13           dbms_output.put_line('sum of '||mgr_id||' is '||count_num);
 14      end if;
 15* end;
```

在上例中的粗体字部分，直接使用 RAISE_APPLICATION_ERROR 来处理错误，在该过程中，我们给出的错误号为-20001，错误消息说明 mgr_id 不能为负值。下面执行该过程，查看一下 RAISE_APPLICATION_ERROR 过程是如何工作的。

```
SQL> /
输入 id 的值:  -122
原值    2:      mgr_id   number(6)  := '&id';
新值    2:      mgr_id   number(6)  := '-122';
declare
*
第 1 行出现错误:
ORA-20001:  error happened...negative mgr_id is not allowed here
ORA-06512: 在 line 7
```

在执行匿名过程时，输入一个负值，此时会抛出 RAISE_APPLICATION_ERROR 定义的异常，并且该异常与 Oracle 的标准异常显示方式一样，有错误号和错误信息提示。但是注意这个错误号是由用户定义的，范围为-20999~-20000，所以，Oracle 支持 1000 个用户定义的使用 RAISE_APPLICATION_ERROR 错误。

显然 RAISE_APPLICATION_ERROR 在用户自定义异常方面更加简洁、方便。除了在可执行语句中使用外，Oracle 也支持在异常处理部分使用这个过程，如实例 12-18 所示。

实例 12-18　在异常处理部分使用 RAISE_APPLICATION_ERROR。

```
SQL> declare
  2     f_name varchar2(20);
  3     l_name varchar2(20);
  4     emp_id  number(6)  := &emp_id;
  5  begin
  6     select first_name,last_name
  7        into f_name,l_name from employees
  8        where employee_id=emp_id;
  9     dbms_output.put_line('first_name is: '||f_name);
 10     dbms_output.put_line('last_name is: ' ||l_name);
 11
 12     exception
 13         when   no_data_found then
 14           raise_application_error (-20002,'This employee does not exist!');
 15* end;
```

上例中的粗体部分为处理异常 NO_DATA_FOUND 的部分，可以使用任何有效地语句来完成对该异常的处理，此时使用了 RAISE_APPLICATION_ERROR 过程来定义异常，使用这个过程就可以在异常发生时，使用与 Oracle 错误一致的方式来得到错误提示了。

但是用户需要维护自定义错误号与错误消息的信息，如实例 12-19 所示，我们创建一个表，用于存储用户自定义异常号与异常描述的含义，这样就可以维护这种错误号与错误消息之间的对应关系，从而不会引起混淆。

实例 12-19　创建一个表 user_def_errors。

```
SQL> create table user_def_errors
  2     (error_number number,
  3      error_message varchar2(1024)
  4     );

表已创建。
```

在上例中已经创建了一个表 user_def_errors，使用这个表可以存储用户定义的错误号与错误消息之间的匹配。下面插入刚才定义的用户错误号以及错误消息，如实例 12-20 所示。

实例 12-20　插入用户定义的用户错误号以及错误消息。

```
SQL> insert into user_def_errors
  2   values (-20002,'This employee does not exist!');

已创建 1 行。

SQL> commit;

提交完成。
```

下面查看一下自定义的错误号与错误消息匹配关系。

```
SQL> col error_message for a50
SQL> set line 120
SQL> select * from user_def_errors

ERROR_NUMBER ERROR_MESSAGE
------------ --------------------------------------------------
      -20002 This employee does not exist!
```

用户可以自己维护这个表，在需要时可以增加或查询更详细的信息，以防止创建重复的用户自定义错误号与错误消息。

12.7　应用EXCEPTION_INIT

使用 EXCEPTION_INIT 定义 Oracle 错误，可以把某个 Oracle 错误号和用户定义错误的名称关联起来。因为有些 Oracle 异常存在错误号，但是没有错误名称，那么在程序员预测到会发生这种错误时，就无法捕获已经知道的异常。使用 EXCEPTION_INIT 编译指令可以完成这个任务，将错误号与用户定义的异常关联起来，再发生这类异常时，就可以使用与错误号关联的异常来捕获这个异常，并加以处理。下面先看一下实例 12-21，在该实例中将发生违反完整性约束的错误，但是该错误没有定义。

实例 12-21　执行匿名过程。

```
SQL> declare
  2      t_deptno number(2) := &deno;
  3  begin
  4      delete from dept
  5         where deptno=t_deptno;
  6      dbms_output.put_line('deleted...1');
  7  end;
  8  /
输入 deno 的值:  20
原值    2:     t_deptno number(2)  := &deno;
新值    2:     t_deptno number(2)  := 20;
declare
*
第 1 行出现错误:
ORA-02292: 违反完整约束条件 (SCOTT.FK_DEPTNO) - 已找到子记录
ORA-06512: 在 line 4
```

在上例中，表 emp 中利用外键引用表 dept 中的列 deptno，当删除表 dept 中的记录时，因为表

emp 中存在参考引用预删除的表 dept 中的记录，所以发生“违反完整约束条件”的错误，该错误号为 ORA-02292。但是该异常没有名称。下面我们使用 EXCEPTION_INIT 编译指令来编译，使得该错误号与用户定义异常结合起来。其语法格式如实例 12-22 所示。

实例 12-22　EXCEPTION_INIT 使用语法。

```
Declare
  Exception_name exception;
  Pragma  exception_init(exception_name,exception_number);
```

下面，我们通过一个具体的实例 12-23 演示如何使用 EXCEPTION_INIT 来完成错误号与用户定义异常的匹配。

实例 12-23　使用 EXCEPTION_INIT 来完成错误号与用户定义异常的匹配。

```
SQL> declare
  2     t_deptno number(2)  := &deno;
  3       constraint_exp exception;
  4       pragma  exception_init(constraint_exp,-2292);
  5  begin
  6     delete from dept
  7        where deptno=t_deptno;
  8     dbms_output.put_line('deleted...');
  9     exception
 10      when constraint_exp then
 11        dbms_output.put_line('violate reference constraint');
 12  end;
 13  /
输入 deno 的值:  20
原值    2:     t_deptno number(2)  := &deno;
新值    2:     t_deptno number(2)  := 20;
violate reference constraint

PL/SQL 过程已成功完成。
```

在上例的 3、4 行定义了用户异常，并且将错误号-2292 与用户定义异常 constraint_exp 关联起来。

12.8　应用SQLCODE与SQLERRM

在程序运行时，总会遇到这样或那样的异常，有些异常是可以预测的，但是依然有些异常是难以确定的，在 PL/SQL 中可以使用 OTHERS 异常处理所有不确定的异常，但是此时就无法知道这个异常的具体内容，从而减少了异常处理的针对性。Oracle 提供了两个函数 SQLCODE 和 SQLERRM 来获得在 OTHERS 异常中捕获的错误号和错误消息，其中 SQLCODE 获得错误号，SQLERRM 获得错误消息，消息的最大长度为 512 个字节。如实例 12-24 所示，编写一个使用 OTHERS 异常的过程，然后再修改这个实例，使用 SQLCODE 与 SQLERRM 函数将获得更有价值的异常信息。

实例 12-24　使用 OTHERS 异常的过程。

```
SQL> declare
  2    var_fname  varchar2(20);
```

```
  3   var_lname  varchar2(20);
  4   emp_date   date;
  5   var_empid  number :='&employee_id';
  6  begin
  7   select first_name,last_name,hire_date
  8    into   var_fname,var_lname,emp_date
  9    from employees
 10    where employee_id=var_empid;
 11  dbms_output.put_line('var_fname is '||var_fname);
 12  dbms_output.put_line('var_lname is '||var_lname);
 13  dbms_output.put_line('var_date is '||emp_date);
 14
 15  exception
 16    when others then
 17     dbms_output.put_line('error happened...');
 18  end;
 19*
```

在本例中，我们使用了 OTHERS 异常来处理程序逻辑可能发生的错误，一旦发生异常，程序逻辑将跳转到 EXCEPTION 处，通过异常处理语句进行处理，之后程序逻辑继续执行。

下面我们执行这个过程，首先输入一个值 89。

```
SQL> /
Enter value for employee_id: 89
old   5:   var_empid  number :='&employee_id';
new   5:   var_empid  number :='89';
error happened...

PL/SQL procedure successfully completed.
```

显然，由于该员工号不存在，所以程序报错，但是具体错误是什么却无从可知。下面我们输入一个负值。

```
SQL> /
Enter value for employee_id: -100
old   5:   var_empid  number :='&employee_id';
new   5:   var_empid  number :='-100';
error happened...

PL/SQL procedure successfully completed.
```

在上例中，我们输入了不存在的员工号，将提示同样的错误，依然没有具体的错误提示。下面我们改写这个实例，在 OTHERS 异常部分使用 SQLCODE 和 SQLERRM 函数获得错误的具体信息，如实例 12-25 所示。

实例 12-25　使用 SQLCODE 和 SQLERRM 函数获得错误的具体信息。

```
SQL> declare
  2   var_fname  varchar2(20);
  3   var_lname  varchar2(20);
  4   emp_date   date;
  5   var_code   number;
  6   var_msg    varchar2(200);
  7   var_empid  number :='&employee_id';
  8  begin
  9   select first_name,last_name,hire_date
```

```
10    into   var_fname,var_lname,emp_date
11    from employees
12    where employee_id=var_empid;
13  dbms_output.put_line('var_fname is '||var_fname);
14  dbms_output.put_line('var_lname is '||var_lname);
15  dbms_output.put_line('var_date is '||emp_date);
16  exception
17    when others then
18     var_code := sqlcode;
19     var_msg  := sqlerrm;
20   dbms_output.put_line('error code :'||var_code);
21   dbms_output.put_line('error message :'||var_msg);
22* end;
```

上例中的粗体部分是涉及的修改内容，首先在声明中定义两个变量 var_code 与 var_msg，用于存储在异常中获得的异常错误号和异常消息。然后在异常处理语句中，将获得的异常错误号和错误消息存储到变量 var_code 与 var_msg 中，然后打印这个错误消息，下面执行这个过程。

```
SQL> /
Enter value for employee_id: 89
old   7:   var_empid  number :='&employee_id';
new   7:   var_empid  number :='89';
error code :100
error message :ORA-01403: no data found

PL/SQL procedure successfully completed.
```

在上例中输入了员工号 89，此时发生异常，错误号为 ORA-01403，该异常是 Oracle 预定义的错误。下面修改该程序，再看一下异常错误信息是什么。修改内容如下所示。

```
declare
  var_fname  varchar2(3);
  var_lname  varchar2(3);
  emp_date   date;
  var_code   number;
  var_msg    varchar2(200);
  var_empid  number :='&employee_id';
```

此时对变量 var_fname 的变量参数做了修改，由 20 改为 3。下面继续执行该过程。

```
SQL> /
Enter value for employee_id: 100
old   7:   var_empid  number :='&employee_id';
new   7:   var_empid  number :='100';
error code :-6502
error message :ORA-06502: PL/SQL: numeric or value error: character string
buffer too small

PL/SQL procedure successfully completed.
```

在这个实例中输入了员工号 100，该员工信息是存在的，但是依然发生了异常，这个错误消息是 string buffer too small。显然，在 OTHERS 异常中，使用 SQLCODE 和 SQLERRM 函数可以获得异常的具体信息。

对于 SQLCODE 和 SQLERRM 而言，下面是需要注意的问题：对于 SQLCODE 函数如果没有异常发生，该函数的返回值为 0，而 SQLERRM 则返回 normal 提示，如实例 12-26 所示。

实例 12-26　SQLCODE 和 SQLERRM 需要注意的问题。

```
SQL> l
  1     begin
  2       dbms_output.put_line ('Error code : '|| sqlcode);
  3       dbms_output.put_line ('Error code : '|| sqlerrm);
  4       dbms_output.put_line ('Error code : '|| sqlerrm(-20000));
  5       dbms_output.put_line ('Error code : '|| sqlerrm(200));
  6       dbms_output.put_line ('Error code : '|| sqlerrm(-2292));
  7*    end;
SQL> /
Error code : 0
Error code : ORA-0000: normal, successful completion
Error code : ORA-20000:
Error code :  -200: non-ORACLE exception
Error code : ORA-02292: integrity constraint (.) violated - child record found

PL/SQL procedure successfully completed.。
```

在上例中对于没有异常发生的情况，SQLCODE 和 SQLERRM 都返回了预期的结果，对于 SQLERRM 函数，其参数为错误号，如果输入了不存在的错误号，该函数会提示这个问题，如果输入了预定义的错误号，则输出与该错误号关联的预定义的错误信息，如在上例中函数 SQLERRM(-2292)就是 Oracle 预定义的错误号，返回了预定义的错误信息。

12.9　本章小结

异常是任何编程语言都必须面对的问题，Oracle 将异常分为用户自定义异常和预定义异常，使用预定义异常可以处理大家经常遇到的异常，如除 0 错误、no_data_found 错误等，使用用户自定义异常可以显著增加用户程序的灵活性和健壮性。理解了异常的这些概念后，本章又详细介绍了异常的传播过程，针对在可执行部分、声明部分和异常处理部分发生的异常各自分析了异常的传播过程和异常的处理方法。最后介绍了过程 RAISE_APPLICATION_ERROR、编译过程 EXCEPTION_INIT 以及函数 SQLCODE 和 SQLERRM，利用它们可以使得异常处理更简洁，使得用户在完成错误号与错误信息的匹配后，以进一步确认错误类型。

第 13 章
记　录

PL/SQL 支持三种类型的记录：基于表的记录、基于游标的记录和用户自定义的记录。本章将学习这三种记录类型。记录用来标识一个逻辑实体，例如一个员工，应包含其名称、性别、入职时间、工资、所属部门、员工号等信息，通过这些属性信息来表示一个员工实体，使用记录可以很容易地标识员工这个逻辑单元。

13.1　基于表的记录

基于表的记录可使用某一个表的所有列的属性项作为一个逻辑单元，此时需要使用表的%rowtype 属性，这样我们就简化了对记录的自定义，在实际编程中往往需要依据表的行记录来实现数据操作，如实例 13-1 所示，创建并使用基于表的记录。

实例 13-1　创建并使用基于表的记录。

```
SQL> set serveroutput on;
SQL> declare
  2    employee_rec employees%ROWTYPE; --调用表的%ROWTYPE 属性
  3  begin
  4  select * into employee_rec
  5  from employees
  6  where     employee_id=206;
  7  dbms_output.put_line('employee_id: '||employee_rec.employee_id);
  8  dbms_output.put_line('employee_first_name: '||employee_rec.first_name);
  9 * end;
SQL> /
employee_id: 206
employee_first_name: William
```

在上例中的第 2 行代码声明了一个基于表 employees 的记录，该记录名为 employee_rec，紧接着使用 SELECT 语句填充记录，然后通过 DBMS_OUTPUT.PUT_LINE 过程打印记录对应的属性项的数值，即打印记录的 employee_id 和 first_name。通过执行结果可以验证记录中的数据。

13.2　基于游标的记录

基于游标的记录，用于声明记录与游标具有相同的结构，此时需要首先声明游标，然后声明基于游标的记录，通过 LOOP 循环不断地填充记录，并处理记录中的数据，实例 13-2 将演示如何

创建基于游标的记录，并使用该记录。

实例 13-2 创建基于游标的记录。

```
SQL> set serveroutput on;
SQL> declare
  2    cursor employee_cur is                          --首先声明一个游标
  3        select first_name,last_name,email,job_id
  4          from employees
  5        where rownum<3;
  6    employee_rec employee_cur%ROWTYPE;        --接着声明基于游标的记录
  7  begin
  8   open employee_cur;
  9   loop
 10     fetch employee_cur into employee_rec;            --向记录填充数据
 11     exit when  employee_cur%notfound;
 12
 13     dbms_output.put_line('name : '||employee_rec.first_name);--记录的操作
 14  end loop;
 15* end;
SQL> /
name : Donald
name : Douglas

PL/SQL 过程已成功完成。
```

在上例中，我们首先声明了游标 employee_cur，并将其与 SELECT 语句关联，然后声明一个基于游标的记录 employee_rec。在匿名过程的 begin…end 可执行部分，使用 LOOP 循环来填充记录，并在每次循环时对记录进行操作，这里是打印记录的相关数据项。



使用基于游标的记录必须先声明游标，在声明基于游标的记录时同样使用了%ROWTYPE 属性，这点和基于表的记录的定义类似。

13.3 用户自定义的记录

使用基于表的记录和使用基于游标的记录，二者都是基于已经有的数据类型集合，显然这种记录定义局限于具体的表和具体的游标定义，为了更灵活地使用记录，Oracle 允许用户自定义记录，如实例 13-3 所示。

实例 13-3 创建用户自定义记录。

```
SQL> declare
  2   type time_rec_type is record
  3      (curr_date date,
  4       curr_day  varchar2(12) ,
  5       curr_time varchar2(8) not null :='00:00:00' );
  6     time_rec time_rec_type;
  7   begin
  8    select sysdate
  9      into time_rec.curr_date
 10      from dual;
 11    time_rec.curr_day := to_char(time_rec.curr_date,'DAY');
```

```
 12    time_rec.curr_time:= to_char(time_rec.curr_date,'hh24:mi:ss');
 13
 14    dbms_output.put_line('Date: ' ||time_rec.curr_date);
 15    dbms_output.put_line('Day:  ' ||time_rec.curr_day);
 16    dbms_output.put_line('Time: ' ||time_rec.curr_time);
 17  end;
 18  /
Date: 21-1月 -12
Day:  星期六
Time: 013:41:01

PL/SQL 过程已成功完成。
```

在上例中自定义了一个记录类型 time_rec_type，其具体定义如下所示。

```
  2   type time_rec_type is record
  3      (curr_date date,
  4       curr_day  varchar2(12) ,
  5       curr_time varchar2(8) not null :='00:00:00' );
  6     time_rec time_rec_type;
```

其中 time_rec_type 是记录的类型，而在括号中的部分是记录的属性，time_rec 是基于自定义记录类型的记录名称，也就是可以操作的记录对象名称。

在匿名过程的 begin…end 可执行部分为记录填充数据，并使用 dbms_output.put_line 过程打印记录的所有数据项。

在用户自定义记录中，如果记录的非空字段声明为 NOT NULL，此时必须为字段赋予初值，否则会报错，如实例 13-4 所示。

实例 13-4　非空字段没有赋予初值的错误。

```
SQL> declare
  2   type time_rec_type is record
  3      (curr_date date,
  4       curr_day  varchar2(12) ,
  5       curr_time varchar2(8) not null);  --此处的非空字段没有赋予初值。
  6     time_rec time_rec_type;
  7   begin
  8    select sysdate
  9      into time_rec.curr_date
 10      from dual;
 11    time_rec.curr_day := to_char(time_rec.curr_date,'DAY');
 12    time_rec.curr_time:= to_char(time_rec.curr_date,'hh24:mi:ss');
 13
 14    dbms_output.put_line('Date: ' ||time_rec.curr_date);
 15    dbms_output.put_line('Day:  ' ||time_rec.curr_day);
 16    dbms_output.put_line('Time: ' ||time_rec.curr_time);
 17  end;
 18  /
     curr_time varchar2(8) not null);
     *
第 5 行出现错误:
ORA-06550: 第 5 行, 第 7 列:
PLS-00218: 声明为 NOT NULL 的变量必须有初始化赋值
```

13.4 嵌套记录

记录可以理解为一种数据类型，在记录的定义中，自然可以使用记录的嵌套，即在记录中有记录。

实例 13-5 为定义嵌套记录的部分代码。

实例 13-5 嵌套记录。

```
declare
  type rec_type is record
    (
     first_name varchar2(20),
     last_name  varchar2(25)
    );

  type emp_type is record
    (name           rec_type,
     email          varchar2(25),
     hire_date      date,
     salary         number(8,2),
     department_id  number(4)
     );

  emp_rec  emp_type;
```

在上例中，首先声明了一个用户自定义记录 rec_type，然后又声明了一个用户自定义记录 emp_type，但是在该记录的属性中，name 的数据类型为用户自定义记录 rec_type。这种记录称为记录的嵌套。下面是一个具体的实例，如实例 13-6 所示。

实例 13-6 创建嵌套记录。

```
SQL> declare
  2   type rec_type is record
  3     (
  4      first_name varchar2(20),
  5      last_name  varchar2(25)
  6     );
  7
  8   type emp_type is record
  9     (name           rec_type,
 10      email          varchar2(25),
 11      hire_date      date,
 12      salary         number(8,2),
 13      department_id  number(4)
 14      );
 15
 16   emp_rec  emp_type;
 17
 18   begin
 19    select first_name,last_name,email,hire_date,salary,department_id
 20      into emp_rec.name.first_name,emp_rec.name.last_name,
 21    emp_rec.email,emp_rec.hire_date,emp_rec.salary,emp_rec.department_id
       from employees where employee_id=206;
 22
```

```
  23    dbms_output.put_line ('name: '||emp_rec.name.first_name||'
        '||emp_rec.name.last_name);
  24    dbms_output.put_line('hire_date is : '||emp_rec.hire_date);
  25    dbms_output.put_line('email is : '||emp_rec.email);
  26    dbms_output.put_line('salary is : '||emp_rec.salary);
  27    dbms_output.put_line('department_id is : '||emp_rec.department_id);
  28* end;
```

在上例中定义了一个嵌套记录类型 emp_type，并声明了该记录类型的一个记录 emp_rec，通过 SELECT 语句为该嵌套记录赋值，此时使用了多层调用为嵌套记录中的记录类型的属性赋值，如 emp_rec.name.first_name，最后通过打印嵌套记录的属性值验证填充记录的结果。下面是执行上述匿名过程的结果集。

```
SQL> /
name: William Gietz
hire_date is : 07-6月 -134
email is : WGIETZ
salary is : 8300
department_id is : 110

PL/SQL 过程已成功完成。
```

13.5　记录集合

记录作为一个单独的逻辑单元同样可以组成一个集合，即记录的集合，在该集合中每一个下标的位置对应一个记录，通过下标以及记录的属性访问记录中的数据对象，如实例 13-7 所示。

实例 13-7　创建记录的集合。

```
SQL> declare
  2   cursor emp_cur is
  3     select first_name,last_name
  4     from employees
  5      where rownum<=3;
  6   type emp_type is table of emp_cur%ROWTYPE –定义联合数组类型
  7   index by binary_integer;
  8
  9   emp_tab  emp_type; --定义联合数组。
 10
 11   var_counter integer :=0;
 12
 13 begin
 14   for i in emp_cur loop  --通过 LOOP 循环为联合数组填充记录。
 15      var_counter := var_counter+1;
 16
 17      emp_tab(var_counter).first_name := i.first_name;
 18      emp_tab(var_counter).last_name  := i.last_name;
 19
 20 dbms_output.put_line ('tab first_name is : '||var_counter||emp_tab
 (var_counter).firs
 21      dbms_output.put_line ('tab last_name is  : '||var_counter||emp_tab
 22      (var_counter).last_name);
 23   end loop;
 24 end;
```

在上例中首先声明一个游标，该游标返回 3 行记录，然后声明一个数组类型，该数组类型是游标的%rowtype 属性，然后定义一个数组变量和一个计数器，用来具体操作数组中的记录数据。如下代码用于向联合数组填充记录。

```
        emp_tab(var_counter).first_name := i.first_name;
        emp_tab(var_counter).last_name  := i.last_name;
```

以上代码在联合数组的每一个下标位置存储了一个记录。下面执行该记录，执行结果如下所示。

```
    tab first_name is : 1Ellen
    tab last_name is  : 1Abel
    tab first_name is : 2Sundar
    tab last_name is  : 2Ande
    tab first_name is : 3Mozhe
    tab last_name is  : 3Atkinson

    PL/SQL 过程已成功完成。
```

13.6 本章小结

本章介绍了各种类型的记录，即基于表的记录、基于游标的记录和用户自定义的记录，同时介绍了嵌套记录和记录的集合，更加丰富了使用记录编程的客观需求。

第 14 章

集合类型

集合类型是任何高级编程语言中都必须具有的一种数据类型，Oracle 的 PL/SQL 编程语言也提供了几种集合类型，以供用户选择，这些类型包括联合数组、嵌套表、变长数组以及多层集合，最后还介绍了集合操作的方法，通过本章的学习，尤其是对大量示例的学习和使用，相信读者可以轻松掌握 PL/SQL 的集合使用方法。

14.1　联合数组

联合数组也叫索引表，用于存储某个数据类型的数据的集合类型，可以通过索引获得联合数组中的数据。下面是创建联合数组的语法。

```
type type_name is table of element_type [not null]
 index by element_type;
 table_name type_name;
```

在联合数组的语法结构中，可使用 type 声明表结构，并对结构进行说明。下面通过实例 14-1 演示如何创建联合数组。

实例 14-1　演示联合数组的用法。

```
SQL> declare
  2   cursor my_cursor is
  3     select ename
  4       from emp
  5      where sal<=2800;
  6  type ename_type is table of emp.ename%type –声明联合数组类型
  7     index by binary_integer;                  --声明联合数组的索引
  8     ename_table ename_type;       --声明 ename_type 的联合数组 ename_table
  9      var_counter integer := 0;
 10  begin
 11    for ename_rec in my_cursor loop
 12     var_counter := var_counter+1;
 13  ename_table(var_counter) := ename_rec.ename;--循环游标将数据存入联合数组。
 14     dbms_output.put_line ('ename('||var_counter||'):' ||ename_table(var_
      counter));
 15  end loop;
 16  end;
 17  /
ename(1):  SMITH
ename(2):  ALLEN
```

```
ename(3): WARD
ename(4): MARTIN
ename(5): CLARK
ename(6): TURNER
ename(7): ADAMS
ename(8): JAMES
ename(9): MILLER

PL/SQL 过程已成功完成。
```

第 6 行用于声明联合数组所使用的类型，与表 emp 的列 ename 的类型相同，第 8 行用于声明联合数组的名称为 ename_table，其类型为 ename_type，第 13 行使用下标 var_counter 来引用单独的联合数组位置，用于填充数据。

14.2 嵌套表

嵌套表也是 PL/SQL 表的类型之一，它与联合数组具有相同的结构，都是使用下标访问数据，二者的主要区别是嵌套表可以存储在数据库表的列中，而联合数组却不能。下面是嵌套表的语法结构。

```
Type type_name  is table of element_type [not null];
 Table_name type_name;
```

这里要注意嵌套表的语法结构与声明联合数组的区别，使用嵌套表必须先初始化表，否则会报错，如实例 14-2 所示。

实例 14-2　创建嵌套表（未初始化）。

```
SQL> declare
  2    cursor my_cursor is
  3      select ename
  4        from emp
  5       where sal<=2800;
  6   type ename_type is table of emp.ename%type;
  7       ename_table ename_type ;   --该表没有初始化，运行时会报错。
  8      var_counter integer := 0;
  9   begin
 10    for ename_rec in my_cursor loop
 11     var_counter := var_counter+1;
 12     ename_table.extend;
 13     ename_table(var_counter) := ename_rec.ename;
 14   dbms_output.put_line ('ename('||var_counter||'):'||ename_table(var_counter);
 15   end loop;
 16   end;
 17   /
declare
*
第 1 行出现错误:
ORA-06531: 引用未初始化的集合
ORA-06512: 在 line 12
```

上例的错误输出说明，在使用集合类型嵌套表时，在声明嵌套之后，必须首先初始化嵌套表，然后再使用，否则会报错，下面就是实现嵌套表初始化后的实例，如实例 14-3 所示。

实例 14-3　使用嵌套表（初始化）。

```
SQL> declare
  2   cursor my_cursor is
  3     select ename
  4       from emp
  5      where sal<=2800;
  6  type ename_type is table of emp.ename%type;  --声明嵌套表的元素类型
  7     ename_table ename_type := ename_type(); --嵌套表必须先初始化才能使用。
  8     var_counter integer := 0;
  9  begin
 10   for ename_rec in my_cursor loop
 11    var_counter := var_counter+1;
 12    ename_table.extend;                    --调用 EXTEND 方法增加集合的大小。
 13    ename_table(var_counter) := ename_rec.ename;
 14    dbms_output.put_line ('ename('||var_counter||'):  ' ||ename_table
      (var_counter)
);
 15  end loop;
 16  end;
 17  /
ename(1):  SMITH
ename(2):  ALLEN
ename(3):  WARD
ename(4):  MARTIN
ename(5):  CLARK
ename(6):  TURNER
ename(7):  ADAMS
ename(8):  JAMES
ename(9):  MILLER

PL/SQL 过程已成功完成。
```

请注意第 7 行，第 7 行用于初始化嵌套表，这里产生一个空值，但不是 NULL 值，从而完成嵌套表的初始化，还可以避免盲目设置嵌套表的初始值无法确定的问题。

14.3　变长数组

变长数组是一种集合类型，变长数组为每个元素分配一个从 1 开始计数的下标，对应变长数组的位置。变长数组的尺寸受定义变长数组时的限制，元素数量从 0 到最大值之间，如图 14-1 所示。

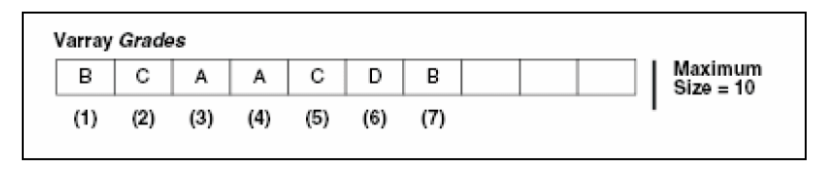


图 14-1　变长数组

变长数组的定义如下所示。

```
type type_name  is { varry(n) | varrying array(n) }of element_type ;
  varry_name  type_name;
```

type_name 是变长数组的类型名称，变长数组有两种类型：varry(n)和 varrying array(n)，数值 n 用于指定变长数组的长度，即元素的数量。

下面我们根据上图创建一个变长数组，如实例 14-4 所示。

实例 14-4　创建变长数组。

```
SQL> declare
  2   type var_array_type is varray(10) of number(10);
  3   varray_array var_array_type := var_array_type();
  4    var_counter integer :=0;
  5  begin
  6   for i in 1..7 loop
  7     var_counter := var_counter + 1;
  8      varray_array.extend;
  9      varray_array(var_counter) := i;
 10      dbms_output.put_line('varray_array('||i||')=='||varray_array(i));
 11   end loop;
 12  end;
 13  /
```

在上例中，我们定义了一个变长数组 varray_array，数组类型为 varray，长度为 10，插入的数据类型为 NUMBER。上述过程定义了一个计数器，通过计数器以及一个 LOOP 循环向变长数组填充数据。下面是匿名过程的执行结果。

```
varray_array(1)==1
varray_array(2)==2
varray_array(3)==3
varray_array(4)==4
varray_array(5)==5
varray_array(6)==6
varray_array(7)==7

PL/SQL 过程已成功完成。
```

上例通过变长数组的下标访问变长数组中的数据并打印。在上例中，我们定义了变长数组的长度是 10，并填充了从 1~7 位置处的数据，那么还有三个位置可以填充数据，在没有填充数据前 varray_array(8)以及以后的两个位置存放什么数据呢？下面我们验证一下，如实例 14-5 所示。

实例 14-5　验证变长数组。

```
SQL> declare
  2   type var_array_type is varray(10) of number(10);
  3   varray_array var_array_type := var_array_type();
  4    var_counter integer :=0;
  5  begin
  6   for i in 1..7 loop
  7     var_counter := var_counter + 1;
  8      varray_array.extend;
  9      varray_array(var_counter) := i;
 10      dbms_output.put_line('varray_array('||i||')=='||varray_array(i));
 11   end loop;
 12
 13  for i in 1..10 loop
 14    dbms_output.put_line(varray_array(i));
 15    dbms_output.put_line('-----------');
```

```
 16  end loop;
 17  end;
 18  /
varray_array(1)==1
varray_array(2)==2
varray_array(3)==3
varray_array(4)==4
varray_array(5)==5
varray_array(6)==6
varray_array(7)==7
1
-----------
2
-----------
3
-----------
4
-----------
5
-----------
6
-----------
7
-----------
declare
*
第 1 行出现错误:
ORA-06533: 下标超出数量
ORA-06512: 在 line 14
```

在上例中，变长数组 varray_array 存储了 7 个数据，可以通过下标进行访问，而下标为 8 的位置则无法访问，显示“下标超出数量”的错误，说明有效的下标为 7 个，所以如果需要继续向变长数组填充数据，需要使用集合函数 EXTEND 来扩展集合空间，然后填充数据，而后这些位置才可以访问，修改代码如实例 14-6 所示。

实例 14-6　修改代码。

```
SQL> declare
  2   type var_array_type is varray(10) of number(10);
  3   varray_array var_array_type := var_array_type();
  4    var_counter integer :=0;
  5  begin
  6   for i in 1..7 loop
  7     var_counter := var_counter + 1;
  8      varray_array.extend;
  9      varray_array(var_counter) := i;
 10   end loop;
 11       varray_array.extend;
 12       varray_array(8)  := 10;
 13       dbms_output.put_line('------------------------');
 14      dbms_output.put_line('varray_array(8)=='||varray_array(8));
 15   dbms_output.put_line('varray_array.count =='||varray_array.count);
 16  end;
 17  /
```

在上例中的第 11~15 行，首先通过 EXTEND 函数扩展集合空间，然后为下标为 8 的位置赋值，

最后通过 DBMS_OUTPUT 来打印该位置的数据，并计算当前变长数组的元素数量，执行结果如下所示。

```
------------------------
varray_array(8)==10
varray_array.count ==8

PL/SQL 过程已成功完成。
```

从执行结果可以知道，已成功为变长数组 varray_array 赋值，并且当前的元素数量为 8，而不是 10。变长数组集合函数的使用如实例 14-7 所示。

实例 14-7 变长数组中的集合函数使用。

```
SQL> declare
  2   type var_array_type is varray(10) of number(10);
  3   varray_array var_array_type := var_array_type();
  4    var_counter integer :=0;
  5  begin
  6   for i in 1..7 loop
  7     var_counter := var_counter + 1;
  8      varray_array.extend;
  9      varray_array(var_counter) := i;
 10   end loop;
 11   dbms_output.put_line('varray_array.count ='|| varray_array.count);
     --元素数量
 12   dbms_output.put_line('varray_array.first ='|| varray_array.first);
     --第一个下标
 13   dbms_output.put_line('varray_array.last  ='|| varray_array.last);
     --最后一个下标
 14   varray_array.extend(3,6);
 15  dbms_output.put_line('varray_array(8)  ='||varray_array(8));
 16  dbms_output.put_line('varray_array(9)  = '||varray_array(9));
 17  dbms_output.put_line('varray_array(10)='||varray_array(10));
 18   dbms_output.put_line('varray_array.last ='|| varray_array.last);
     --最后一个下标
 19
 20   varray_array.trim(4);                          -- 删除后 4 个元素以及下标。
 21   dbms_output.put_line('varray_array.last ='|| varray_array.last);
     --删除 4 个元素后的最后一个下标值。
 22  end;
 23  /
```

输出如下所示。

```
varray_array.count =7
varray_array.first =1
varray_array.last  =7
varray_array(8) =6
varray_array(9) = 6
varray_array(10)=6
varray_array.last =10
varray_array.last =6

PL/SQL 过程已成功完成。
```

在上例中，我们调用了集合的各种方法，通过这些方法的使用可体会这些方法的具体作用，

在实例的后面使用注释解释了这些方法的作用，读者只要在数据库中运行一下给出的实例，就很容易理解集合的用法和作用，这里不再赘述。

14.4　多层集合

前面已经介绍了集合的几种类型，这些集合的元素类型往往基于基本数据类型，如 NUMBER 等。Oracle 11g 支持基于集合的集合，即集合作为集合的元素，称为多层集合。

下面先通过一个实例学习如何创建多层集合，在解释这个实例之后，相信读者应该对其具有清晰地理解，然后我们再给出创建多层集合的语法规则。

这个多层集合的元素是变长数组，变长数组的元素类型为 INTEGER。这个实例的多层集合的示意图如图 14-2 所示。

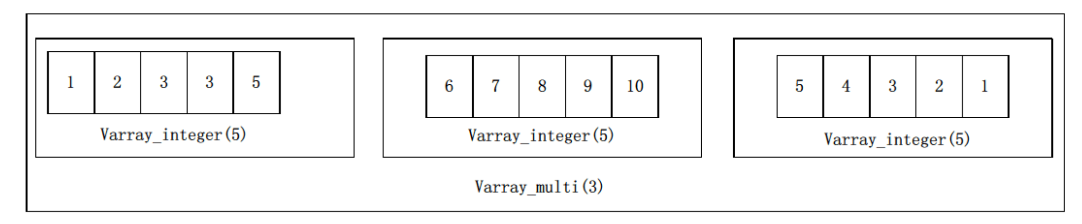


图 14-2　多层集合示意图

下面就基于上图的多层集合的示意图演示如何创建一个多层集合，如实例 14-8 所示。

实例 14-8　创建多层集合。

```
SQL> declare
  2  type var_tye1 is varray(5) of integer;--声明变长数组
  3  type var_tye2 is varray(3) of var_tye1;--声明多层集合，集合元素类型为 var_tye1
  4  varray_integer var_tye1 := var_tye1(1,2,3,4,5);
  5  varray_multi   var_tye2 := var_tye2(varray_integer);
  6
  7  begin
  8    dbms_output.put_line('varray_integer value : ');
  9    for i in 1..5 loop
 10      dbms_output.put_line('varray_integer('||i||')is:'||varray_integer(i));
 11     end loop;
 12
 13   varray_multi.extend;
 14   varray_multi(2)  := var_tye1(6,7,8,9,10);
 15
 16   varray_multi.extend;
 17   varray_multi(3)  := var_tye1(5,4,3,2,1);
 18
 19  for i in 1..3 loop
 20    for j in 1..5 loop
 21     dbms_output.put_line
 22       ('varray_multi['||i||']'||'['||j||'] is :'||varray_multi(i)(j));
 23    end loop;
 24  end loop;
 25  end;
 26  /
```

```
    varray_integer value :
    varray_integer(1) is :1
    varray_integer(2) is :2
    varray_integer(3) is :3
    varray_integer(4) is :4
    varray_integer(5) is :5
    varray_multi[1][1] is :1
    varray_multi[1][2] is :2
    varray_multi[1][3] is :3
    varray_multi[1][4] is :4
    varray_multi[1][5] is :5
    varray_multi[2][1] is :6
    varray_multi[2][2] is :7
    varray_multi[2][3] is :8
    varray_multi[2][4] is :9
    varray_multi[2][5] is :10
    varray_multi[3][1] is :5
    varray_multi[3][2] is :4
    varray_multi[3][3] is :3
    varray_multi[3][4] is :2
    varray_multi[3][5] is :1

    PL/SQL 过程已成功完成。
```

在上例中我们声明了多层集合 varray_multi，然后向该多层集合填充变长数组 varray_integer，最后打印变长数组和多层集合中的数据。

代码的第 2 行和第 3 行用于声明变长数组类型和多层集合类型。

```
     2   type var_tye1 is varray(5) of integer;--声明变长数组
     3   type var_tye2 is varray(3) of var_tye1;--声明多层集合，集合元素类型为 var_tye1
```

第 4 行和第 5 行用于声明变长数组变量、多层集合变量并赋予初值。

```
     4   varray_integer var_tye1 := var_tye1(1,2,3,4,5);
     5   varray_multi   var_tye2 := var_tye2(varray_integer);
```

以上这些部分都属于变量声明或者赋初值部分，在匿名过程的可执行部分，主要是第 10 行，我们打印变长数组中的数据。

```
    10  dbms_output.put_line('varray_integer('||i||') is :'||varray_integer(i));
```

接下来是继续填充多层集合的操作，因为在第 5 行中声明了多层集合并赋予了初值，所以这里继续填充多层集合，如第 13 行和第 14 行所示，先使用集合的 EXTEND 函数扩展多层集合，然后为该集合赋予一个变长数组，第 16 行和第 17 行的功能与其类似。

```
    13   varray_multi.extend;
    14   varray_multi(2)  := var_tye1(6,7,8,9,10);
    16   varray_multi.extend;
    17   varray_multi(3)  := var_tye1(5,4,3,2,1);
```

紧接着使用两层循环遍历多层集合的元素（其实多层集合就如同在高级语言中的多层数组，只不过多层集合的元素类型更加丰富），然后打印这些数据以验证填充结果。

```
    19  for i in 1..3 loop
    20   for j in 1..5 loop
    21    dbms_output.put_line
```

```
 22       ('varray_multi['||i||']'||'['||j||'] is :'||varray_multi(i)(j));
 23    end loop;
 24  end loop;
```

对于多层集合，显然可以使用多种类型的集合来定义并填充，如上例使用了变长数组，也可以利用嵌套表或者变长数组的嵌套表来实现对多层集合的填充。其定义、填充、使用均和上例中介绍的类似。

14.5　集合的方法

集合的方法是内置函数，集合对象可以直接调用，如 EXTEND 方法，可实现增加集合大小的功能，如果没有这个函数，则嵌套表集合将无法扩展。在 Oracle 官方文档中说明的集合方法如图 14-3 所示。

Method	Type	Description
DELETE	Procedure	Deletes elements from collection.
TRIM	Procedure	Deletes elements from end of varray or nested table.
EXTEND	Procedure	Adds elements to end of varray or nested table.
EXISTS	Function	Returns TRUE if and only if specified element of varray or nested table exists.
FIRST	Function	Returns first subscript in collection.
LAST	Function	Returns last subscript in collection.
COUNT	Function	Returns number of elements in collection.
LIMIT	Function	Returns maximum number of elements that collection can have.
PRIOR	Function	Returns subscript that precedes specified subscript.
NEXT	Function	Returns subscript that succeeds specified subscript.

图 14-3　集合方法

对上图中 10 种方法的具体作用说明如下。

- DELETE：删除集合元素。
- EXTEND：为集合增加元素空间，实现空间的扩展以填充新的元素。
- COUNT：集合中元素的数量。
- EXISTS：如果指定的元素在集合中存在，则返回 true。
- DELETE：删除指定集合位置的元素。
- FIRST AND LAST：返回集合的第一个和最后一个元素。
- PRIOR AND NEXT：返回集合指定位置的前一个和后一个元素。
- TRIM：从集合的尾部删除元素，即删除集合的最后一个元素。
- LIMIT：返回集合允许的元素最大数量。

下面我们使用联合数组演示集合内置函数的使用方法，如实例 14-9 所示。

实例 14-9　使用集合内置方法。

```
SQL> declare
  2   type asso_array_type is table of number
  3     index by binary_integer;
  4     asso_array_tab asso_array_type;
  5  begin
```

```
  6  for i in 1..20 loop
  7   asso_array_tab(i) := i;
  8 end loop;
  9  dbms_output.put_line('asso_array_tab.count '||asso_array_tab.count);
 10  asso_array_tab.delete(20);
 11 dbms_output.put_line('asso_array_tab.count '||asso_array_tab.count);
 12 dbms_output.put_line('asso_array_tab.last  '||asso_array_tab.last);
 13  asso_array_tab.delete(1,5);
 14 dbms_output.put_line('asso_array_tab.count '||asso_array_tab.count);
 15 dbms_output.put_line('asso_array_tab.first  '||asso_array_tab.first);
 16  if asso_array_tab.exists(17) then
 17   dbms_output.put_line('asso_array_tab.exists(17)is'||asso_array_ tab(17));
 18  end if;
 19 dbms_output.put_line('asso_array_tab.prior(5) ='||asso_array_tab.prior (5));
 20 dbms_output.put_line('asso_array_tab.next(5) ='||asso_array_tab.next (5));
 21
 22 end;
 23 /
asso_array_tab.count 20
asso_array_tab.count 19
asso_array_tab.last  19
asso_array_tab.count 14
asso_array_tab.first  6
asso_array_tab.exists(17) is 17
asso_array_tab.prior(5) =
asso_array_tab.next(5) =6

PL/SQL 过程已成功完成。
```

下面说明执行结果：在匿名过程中定义了联合数组 asso_array_tab，填充的数据类型为 number，通过一个 FOR 循环将 1~20 的数字填充到联合数组中去。然后调用集合函数，并通过调用包 DBMS_OUTPUT 来打印集合函数的计算结果。由于联合数组的长度就是 FOR 循环的次数，所以第 1 行的输出如下所示。

```
asso_array_tab.count 20
```

第 2 行的输出如下所示，因为在执行这个结果之前的第 10 行调用了 DELETE 函数，删除了联合数组的最后位置的数据，同时删除了占位符。

```
asso_array_tab.count 19
```

而第 3 行用于显示联合数组的最后一个占位符的数据，显然是 19。

```
asso_array_tab.last  19
```

在第 4 行输出执行前，调用 DELETE 函数删除了从 1~5 的联合数组占位符中的数据，所以 COUNT 函数的计算结果是 14。

```
asso_array_tab.count 14
```

第 5 行的输出是计算当前联合数组的第一个占位符中的数值，显然删除了 1~5 占位符的数据，此时第一个占位符的数据就是 asso_array_tab(6)的数值。

```
asso_array_tab.first  6
```

第 6 行用于判断联合数组对应位置的元素是否存在，如果存在，则返回对应位置的值。

```
asso_array_tab.exists(17) is 17
```

第 7 行因为删除了联合数组的 1~5 位置的元素，所以这里的 PRIOR 函数返回 NULL。

```
asso_array_tab.prior(5) =
```

第 8 行返回联合数组位置 5 后的第一个下标的值，所以输出值为 asso_array_tab(6)的值。

```
asso_array_tab.next(5) =6
```

14.6　本章小结

本章介绍了 PL/SQL 的集合类型，在任何编程语言中集合类型都是十分重要和使用方便的数据类型，对于编写复杂的数据集合以及业务关系十分有利。本章介绍了 4 种集合类型，这些类型包括联合数组、嵌套表、变长数组以及多层集合（集合的集合），并通过具体的实例讲解集合类型的定义、使用以及注意事项，最后还介绍了在集合操作中集合方法的使用。

第 15 章

PL/SQL中的SQL

动态 SQL 语句是在运行时，根据输入的参数不同而立即构建对应的 SQL 语句，如 SQL 语句只有在运行时才知道参数，或者 SQL 语句要操作的表只有在 SQL 语句运行时才知道，其特点就是只有在程序运行时才能确定 SQL 语句操纵的对象或者数据，方能基于动态的对象重新构建 SQL 语句。本章将学习静态 SQL 语句的部分内容，但是重点还是如何利用本地动态 SQL 的特性以及如何灵活地使用动态 SQL。

15.1 静态SQL

静态 SQL 是在不同实例中运行都不会变化的 SQL，这种 SQL 语句已经被编译且其存储不会发生变化。本节将介绍静态 SQL 语句的几种使用方式，并通过具体实例给出演示。

15.1.1 在 PL/SQL 中使用 SELECT INTO 初始化变量

对于 PL/SQL 中的变量可以在定义时通过赋值的方式初始化，也可以使用 SELECT INTO 初始化变量。变量一旦被初始化，就可以被过程中的其他元素如过程或者函数使用。下面将演示如何通过 SELECT INTO 来初始化变量，如实例 15-1 所示。

实例 15-1 使用 SELECT INTO 初始化变量。

```
SQL> set serveroutput on;
SQL> declare
  2  var_emp_sum number;
  3  begin
  4   select count(*)
  5     into var_emp_sum   --初始化变量 var_emp_sum
  6      from emp;
  7   dbms_output.put_line
  8      ('the sum of employees is: '||var_emp_sum);
  9    end;
 10  /
the sum of employees is: 15

PL/SQL 过程已成功完成。
```

在上例中，首先声明了变量 var_emp_sum，目的是存储表 emp 中当前员工的数量，并打印该变量。在代码的第 4~6 行，我们通过 SELECT INTO 初始化了变量 var_emp_sum，然后通过

DBMS_OUTPUT 的过程 PUT_LINE 来打印该数值。

15.1.2　在 PL/SQL 中使用 DML 操作

在 PL/SQL 中使用静态 SQL 语句的 DML 操作，是程序员经常使用的方式，如执行 SELECT、UPDATE 以及 DELETE 操作，前提是这些操作的对象已经十分明确。下面通过实例 15-2 演示如何在过程中使用静态 SQL 语句。

实例 15-2　静态 SQL 语句中的 DML 操作。

```
SQL> declare
  2   var_max_sal number;
  3   var_name  varchar2(20);
  4  begin
  5   select max(sal)
  6     into var_max_sal
  7      from emp;
  8   select ename
  9     into var_name
 10      from emp
 11       where sal=var_max_sal;
 12     dbms_output.put_line
 13        ('var_name is :'||var_name);
 14     dbms_output.put_line
 15        ('var_max_sal is :'|| var_max_sal);
 16     delete from emp
 17       where sal=var_max_sal;
 18   end;
 19   /
var_name is :KING
var_max_sal is :5000

PL/SQL 过程已成功完成。
```

在上例的匿名过程中使用了静态 DML 语句，该语句在执行过程中执行：首先声明了两个变量，其中 var_max_sal 用于存储当前表中最大的工资数额，而变量 var_name 用于存储当前工资最多的用户名称，然后在过程的执行部分使用 SELECT INTO 语句来填充这些变量，最后通过一个 DML 操作——DELETE 删除该员工记录。在上例中，执行了该匿名过程，显然过程中的 DML 语句也被执行。

下面在执行上述匿名过程的同一个会话中验证匿名过程的执行结果，如实例 15-3 所示。

实例 15-3　验证 DML 语句的执行结果。

```
SQL> select max(sal) from emp;

  MAX(SAL)
----------
      3000
```

但是有一个问题需要注意，若另起一个会话，继续执行上述查询语句，则可查看的最大 SAL 是多少呢？验证代码如实例 15-4 所示。

实例 15-4　验证代码。

```
C:\Documents and Settings\Administrator>sqlplus scott/oracle

SQL*Plus: Release …… Production
With the Partitioning, OLAP and Data Mining options

SQL> select max(sal) from emp;

  MAX(SAL)
----------
      5000
```

此时，我们发现实际上并没有删除工资最高的员工记录。问题在于实例 15-2 中没有提交 DML 的操作，所以执行完该匿名过程后事务并没有提交，所以新的会话看到的依然是老数据。可以在 DML 语句后使用 COMMIT 语句，也可以在执行匿名过程后在会话窗口中使用 COMMIT 语句，总之只有提交后的事务才会永久更改。

15.2　动态SQL

动态 SQL 语句可以包含有效的 SQL 语句，也可以包含 SQL 语句块以及使用 USING 和 RETURNING INTO 子句，下面我们就介绍这几种动态 SQL 语句的使用方式。

15.2.1　动态 SQL 中包含有效的 SQL 语句

在这个实例中，我们创建一个表，读取部分数据后打印这些数据，最后删除数据。在实际应用中会出现使用中间表的情况，即使用动态 SQL 创建中间表，将数据先汇总到中间表，然后执行逻辑运算，最后删除这张表，如实例 15-5 所示。

实例 15-5　包含有效 SQL 语句的动态 SQL 语句演示。

```
SQL> declare
  2    sql_states varchar2(5000);
  3    counter    number;
  4  begin
  5     sql_states :='create table pid '||
  6     'as select object_id,object_name from dba_objects where rownum<11';
  7     execute immediate sql_states;
  8     execute immediate 'select count(distinct(object_id)) from pid '
  9      into counter;
 10     dbms_output.put_line('counter is '||counter);
 11*  end;

counter is 10

PL/SQL procedure successfully completed.
```

在上例中，执行了两个动态 SQL 语句：一个是将 SQL 语句赋予一个 varchar2 数据类型的变量，然后使用 execute immediate 来执行该语句，创建一个表 PID；另一个 SQL 语句是查询表 PID 中具有不同值的 OBJECT_ID 属性值的数量，此时该语句直接执行，并使用 INTO 语句将返回的单行值

赋予预定义变量 counter。

15.2.2　动态 SQL 中包含 PL/SQL 语句块

使用动态 SQL 语句时，不但可以使用直观的 SQL 执行语句，也可以创建 PL/SQL 语句块，作为一个整体执行，如实例 15-6 所示。

实例 15-6　包含 PL/SQL 语句块的动态 SQL 语句演示。

```
SQL> declare
  2   obj_id    number :=28;
  3   obj_name  varchar2(20);
  4   plsql_blk varchar2(200);
  5  begin
  6    plsql_blk :=' declare var_date date;'  ||
  7              'begin '                 ||
  8              'select sysdate into var_date from dual; ' ||
  9              'dbms_output.put_line(var_date); '||
 10               'end; ';
 11    execute immediate plsql_blk;
 12* end;
SQL> /
13-MAR-12

PL/SQL procedure successfully completed.
```

在上例中，创建了一个简单的 PL/SQL 语句块，该语句块的功能是获取执行该语句块的系统时间，并打印该时间。执行 PL/SQL 语句块与执行普通的动态 SQL 一样，必须使用 EXECUTE IMMEDIATE 语句。

15.2.3　动态 SQL 中使用 USING 和 RETURNING INTO 子句

在动态 SQL 语句中往往需要动态的输入 SQL 语句需要的参数，同时需要输出 SQL 语句操作的数据，将该数据传递给应用程序，此时就可以利用 USING 子句传递数据给动态 SQL 语句，同时使用 RETURNING 子句将数据返回给系统变量，继而传递给应用程序，如实例 15-7 所示。

实例 15-7　使用 USING 与 RETURNING 子句的动态 SQL 语句演示。

```
SQL> declare
  2   obj_id    number :=28;
  3   obj_name  varchar2(20);
  4   name      varchar2(20) := 'mark';
  5   sql_states varchar2(200);
  6  begin
  7   sql_states :='update pid set object_name=:x where object_id =:y '||
  8              'returning object_name into :z';
  9   execute immediate sql_states using name,obj_id returning into obj_name;
 10   dbms_output.put_line('obj_name is '||obj_name);
 11* end;
SQL> /
obj_name is mark

PL/SQL procedure successfully completed.
```

分析上面的实例：在第 7 行和第 8 行，创建动态 SQL 语句，该语句用于更新具有某个 object_id 的 object_name，此时使用了三个绑定变量，这样就使得该 SQL 语句可以共享，从而减少只具有字面值不同的类似 SQL 语句的硬解析，从而提高解析性能，减少 SQL 语句的执行时间。这三个绑定变量一个是“:x”，它对应设置的新的 object_name 的值，一个是“:y”，它用来绑定要修改的 object_id，而绑定变量“:z”用来绑定更新后的值到某个具体的变量。“returning object_name into :z”，用来将更新后的 object_name 的值输出到绑定变量，该绑定变量在执行该 SQL 语句时完成绑定。在第 9 行中，使用了 EXECUTE IMMEDIATE 执行该 SQL 语句，此时，使用了 using 子句向该动态 SQL 语句传递数据，按照顺序分别传递绑定变量“:x”和绑定变量“:y”到具体的变量 object_name 和 obj_id，这两个变量在匿名块的 declare 部分已经定义，然后使用了 returning into 子句将更新后的 object_name 的值传递到该变量 obj_name，同样，该变量也已经在匿名块的 declare 部分定义，最后我们打印更新后的 object_name 的值。

15.2.4 动态 SQL 中使用 EXECUTE IMMEDIATE 的注意事项

通过上面的实例，读者已经对动态 SQL 语句有所了解。使用 EXECUTE IMMEDIATE 执行动态 SQL 语句时，在执行前才能解析动态 SQL 语句，下面是动态 SQL 语句的结构。

```
EXECUTE IMMEDIATE 动态 SQL 语句
[INTO Variable1, Variable2…]
[USING [IN | OUT | IN OUT] 绑定参数 1，绑定参数 2…]
[{RETURNING | RETURN } 属性值 1，属性值 2，… into 绑定参数 1，绑定参数 2…]
```

需要注意动态 SQL 语句与普通 SQL 语句具有一些区别，熟悉这些问题可在以后的编程过程中避免很多错误，从而节约编程时间。下面是读者需要注意的几个问题，同样会通过实例进行演示，从而加深印象。

1. 在 DDL 语句中不能使用绑定变量

在使用动态 SQL 中包含 DDL 语句时，Oracle 不允许使用绑定变量，否则会报错，在数据定义操作中不允许使用绑定变量，如实例 15-8 所示。

实例 15-8 动态 SQL 中 DDL 语句不能使用绑定变量。

```
SQL>declare
  2    sql_states varchar2(5000);
  3    counter    number;
  4    obj_id     number := 50;
  5  begin
  6     sql_states :='create table pid '||
  7     'as select object_id,object_name from dba_objects where object_id= :id';
  8     execute immediate sql_states using obj_id;
  9     execute immediate 'select count(distinct(object_id)) from pid '
 10      into counter;
 11     dbms_output.put_line('counter is '||counter);
 12*  end;
```

在上例的粗体字部分，使用了绑定变量创建表 PID，此时的绑定变量对应于 object_id，也就是希望在执行时再绑定这个值，然后依据这个值来创建表 PID。下面是执行该匿名过程的报错信息。

```
SQL>/
```

```
declare
*
ERROR at line 1:
ORA-01027: bind variables not allowed for data definition operations
ORA-06512: at line 8
```

若要避免这类错误，切记不能在 DDL 语句中使用绑定变量。

2. 动态 SQL 语句的结尾不使用分号（;）

对于静态 SQL 语句需要使用分号（;）作为结束标志，而对于动态 SQL 语句而言，却不能使用分号（；）作为结束标志，否则会报错。下面通过实例 15-9 更直观地演示这个错误。

实例 15-9　动态 SQL 语句结尾使用分号的错误演示。

```
SQL>declare
  2    sql_states varchar2(5000);
  3    counter    number;
  4  begin
  5     execute immediate 'drop table pid';
  6     sql_states :='create table pid '||
  7          'as select object_id,object_name from dba_objects where rownum<11';
  8     execute immediate sql_states;
  9     execute immediate 'select count(distinct(object_id)) from pid ; '
 10      into counter;
 11     dbms_output.put_line('counter is '||counter);
 12*  end;
```

如下是执行该 SQL 语句的结果。

```
SQ  L>/
declare
*
ERROR at line 1:
ORA-00911: invalid character
ORA-06512: at line 9
```

在第 9 行的代码处，我们执行动态 SQL 语句，此时使用了分号（；）作为结束标志，在运行该匿名过程时报错，即在第 9 行处存在无效字符。

3. 动态 SQL 语句的结尾没有右斜线（/）

动态 SQL 语句可以包含有效的 SQL 语句，也可以包含有效的 SQL 语句块，而后通过 EXECUTE IMMEDIATE 来执行该动态 SQL 语句，此时需要注意的是在 SQL 语句块之后不能使用右斜线（/）来标识语句块的结束，否则同样会报错，如实例 15-10 所示。

实例 15-10　动态 SQL 语句块的结尾使用右斜线（/）。

```
SQL>declare
  2   obj_id     number :=28;
  3   obj_name  varchar2(20);
  4   plsql_blk varchar2(200);
  5  begin
  6    plsql_blk :=' declare var_date date;'  ||
  7               'begin '                  ||
  8               'select sysdate into var_date from dual; ' ||
  9               'dbms_output.put_line(var_date); '||
```

```
 10                 'end; /';
 11    execute immediate plsql_blk;
 12* end;
SQL> /
declare
*
ERROR at line 1:
ORA-06550: line 1, column 107:
PLS-00103: Encountered the symbol "/" The symbol "/" was ignored.
ORA-06512: at line 11
```

该错误说明在上述代码的第 11 行执行时遇到 symbol "/"，而这个斜线是在 SQL 语句块中写入的，其初衷是说明执行该语句块，但是在动态 SQL 语句中这个操作是画蛇添足。

4. USING 子句的空值处理

为了学习 USING 子句的空值处理方式，先来看一个实例，这个实例的作用是将表 PID 中 object_id 为 54 的记录的 object_name 属性设置为空值，如实例 15-11 所示。

实例 15-11　动态 SQL 语句中的 USING 子句的空值处理。

```
SQL>declare
  2    sql_states varchar2(200);
  3  begin
  4   sql_states := 'update pid set object_name=:name_null';
  5   execute immediate sql_states using null;
  6  end;
SQL> /
 execute immediate sql_states using null;
                                   *
ERROR at line 5:
ORA-06550: line 5, column 37:
PLS-00457: expressions have to be of SQL types
ORA-06550: line 5, column 2:
PL/SQL: Statement ignored
/
```

上述输出说明，在执行到代码的第 5 行时报错，此时 USING 子句使用了 NULL 作为装配动态 SQL 语句的输入参数，这样的 NULL 值操作是不允许的。下面我们修改上述代码，如实例 15-12 所示。

实例 15-12　修改后的代码。

```
SQL> declare
  2    sql_states varchar2(200);
  3    name       varchar2(20);
  4  begin
  5   sql_states := 'update pid set object_name=:name_null';
  6   execute immediate sql_states
  7   using name;
  8* end;
SQL> /

PL/SQL procedure successfully completed.
```

以上修改了 USING NULL 部分，此时先声明一个 varchar2 类型的变量 name，没有给该变量赋

予初值，此时该变量的值为 NULL，即什么也不是。这个实例说明在使用 USING 子句时，如果需要传入空值，则不能直接使用 NULL，而是需要定义一个变量，再将这个变量传递给 USING 子句。

5. 在 SELECT 语句中表名不能使用绑定变量

当使用动态 SQL 语句时，如果 SELECT 语句中的表名使用绑定变量是不允许的，此时必须使用具体的表名，通过实例 15-13 可更直观地确认这个问题。

实例 15-13　SELECT 语句中表名使用绑定变量的错误演示。

```
SQL>declare
  2    sql_states varchar2(5000);
  3    counter    number;
  4    obj_id     number := 50;
  5  begin
  6     execute immediate 'drop table pid';
  7     sql_states :='create table pid '||
  8          'as select object_id,object_name from dba_objects where rownum<11';
  9     execute immediate sql_states;
 10     execute immediate 'select count(*) from :table_name '
 11      into counter;
 12     dbms_output.put_line('counter is '||counter);
 13   end;
 14*
SQL> /
declare
*
ERROR at line 1:
ORA-00903: invalid table name
ORA-06512: at line 10
```

在上例的第 10 行，表名使用了绑定变量：TABLE_NAME，此时编译报错，错误提示在代码的第 10 行存在无效的表名。

15.3　利用FORALL实现SQL语句的批处理

使用批处理 FORALL 执行 SQL 语句时，可在发送一次 SQL 语句后而执行多次，能够明显提高执行 SQL 语句的效率。下面将演示如何使用 FORALL 实现 SQL 的批处理，因为 SQL 语句与 FORALL 语句一起使用时，会引用集合元素，所以我们会在演示实例中定义集合元素，如实例 15-14 所示。

实例 15-14　演示 FORALL 批处理。

```
SQL>  declare
  2        type type1 is table of number        index by pls_integer;
  3        type type2 is table of varchar2(20)    index by pls_integer;
  4        num_type1          type1;
  5        text_type2           type2;
  6        v_count             number;
  7     begin
  8      for i in 1..6 loop
  9         num_type1(i)  := i;
```

```
 10        text_type2(i) := 'type'||i;
 11        dbms_output.put_line('num_type1(i) : '||num_type1(i) );
 12        dbms_output.put_line('text_type2(i) : '||text_type2(i) );
 13     end loop;
 14    forall i in 1..6
 15         insert into test
 16         values ( num_type1(i) , text_type2(i) );
 17         commit;
 18    select count(*)
 19        into v_count
 20        from test;
 21    dbms_output.put_line('v_count : '||v_count);
 22  end;
 23  /
num_type1(i) : 1
text_type2(i) :  type1
num_type1(i) : 2
text_type2(i) :  type2
num_type1(i) : 3
text_type2(i) :  type3
num_type1(i) : 4
text_type2(i) :  type4
num_type1(i) : 5
text_type2(i) :  type5
num_type1(i) : 6
text_type2(i) :  type6
v_count : 6

PL/SQL 过程已成功完成。
```

在上例中，定义了两个集合元素 num_type1 和 text_type2，首先使用一个 FOR 循环来填充这两个集合，而后使用了 FORALL 实现向表 TEST 中插入数据的批量处理。

前面已经讨论过，使用 FORALL 时需要向 SQL 引擎发送一个 SQL 语句即可执行多次，而使用 FOR 循环则需要发送多次的 SQL 语句，显然从理论上讲使用 FORALL 更加节约时间，下面我们做一个测试，分别使用这两种方式向表中插入同样的数据，查看一下各自的执行时间差异，如实例 15-15 所示。

实例 15-15　测试 FOR 循环与 FORALL 批处理的效率。

```
SQL> declare
  2       type type1 is table of number       index by pls_integer;
  3       type type2 is table of varchar2(20)  index by pls_integer;
  4       num_type1          type1;
  5       text_type2          type2;
  6       var_start           integer;
  7       var_end            integer;
  8       var_dura           number(2,5);
  9     begin
 10     for i in 1..50000 loop
 11        num_type1(i) := i;
 12        text_type2(i) := 'type'||i;
 13     end loop;
 14
 15     var_start := dbms_utility.get_time;
 16    for i in 1..50000 loop
```

```
 17          insert into test
 18           values ( num_type1(i)  , text_type2(i) );
 19           commit;
 20     end loop;
 21     execute immediate 'truncate table test';
 22       var_end := dbms_utility.get_time;
 23       dbms_output.put_line('FOR :'|| (var_end - var_start));
 24
 25       var_start := dbms_utility.get_time;
 26     forall i in 1..50000
 27           insert into test
 28           values ( num_type1(i)  , text_type2(i) );
 29           commit;
 30       var_end := dbms_utility.get_time;
 31    dbms_output.put_line('FORALL :'||(var_end - var_start));
 32  end;
 33  /
FOR :875
FORALL :5

PL/SQL 过程已成功完成。
```

在上例中，我们使用了 FOR 循环和 FORALL 批处理来完成向表 TEST 中插入 50000 行数据，测试二者完成插入操作的时间。此时，我们使用了包 DBMS_UTILITY 的 GET_TIME 函数，该函数可返回精度为 0.01 秒的当前时间，通过插入操作的执行时间差值，测试两种插入数据方式的效率。

从执行结果来看 FOR 循环的执行时间为 875，而 FORALL 批处理的执行时间为 5，显然使用 FORALL 可以极大减少批量 SQL 语句的执行时间。

15.3.1　使用 INDICES OF

使用 INDICES OF 选项可以循环处理稀疏的集合，如将填充满的集合，删除一些数据从而构造稀疏集合，以此测试 INDICES OF 的使用，如实例 15-16 所示。

实例 15-16　使用 INDICES OF 处理稀疏集合。

```
SQL> declare
  2        type type1 is table of number        index by pls_integer;
  3        type type2 is table of varchar2(20)  index by pls_integer;
  4        num_type1         type1;
  5        text_type2         type2;
  6        v_count           number;
  7     begin
  8      for i in 1..6 loop
  9         num_type1(i)  := i;
 10         text_type2(i)  := 'type'||i;
 11      end loop;
 12
 13         text_type2.delete(2);
 14         num_type1.delete(2);
 15         text_type2.delete(4);
 16         num_type1.delete(4);
 17
 18     forall i in  indices of num_type1
 19           insert into test
```

```
 20         values ( num_type1(i)  , text_type2(i) );
 21         commit;
 22     select count(*)
 23         into v_count
 24         from test;
 25     dbms_output.put_line('v_count : '||v_count);
 26  end;
 27  /
v_count : 4

PL/SQL 过程已成功完成。
```

在上例中，我们使用 FOR 循环填充了集合 num_type1 和 text_type2，此时这两个集合都包含 6 个元素，即稀疏集合，从中删除一些数据，即将这两个集合的下标为 2、4 的数据全部删除，然后使用 INDICES OF 处理稀疏矩阵，即只向表 TEST 中插入 4 个值，如实例 15-17 所示。

实例 15-17　查询 TEST 表。

```
SQL>
SQL> select * from test;

   ROW_NUM ROW_TEXT
---------- --------------------
         1 type1
         3 type3
         5 type5
         6 type6

SQL>
```

如果我们不使用 INDICES OF 处理稀疏矩阵而是使用 FORALL IN 的正常形式，则会报错，修改部分代码，如实例 15-18 所示。

实例 15-18　不使用 INDICES OF 处理稀疏集合。

```
18     forall i in 1..6
19          insert into test
20          values ( num_type1(i)  , text_type2(i) );
21          commit;
```

则执行结果会提示如下错误。

```
27  /
declare
*
第 1 行出现错误:
ORA-22150: 下标 [2] 中的元素不存在
ORA-06512: 在 line 18
```

显然按照顺序会先读到集合的第二个元素，此时该位置集合中的元素为 NULL，即没有数据，所以提示下标 2 中的元素不存在，而使用 INDICES OF 就可以处理这种稀疏集合的问题。

15.3.2　使用 VALUES OF

该选项的作用为 FORALL 语句的循环计数器提供一个值，紧跟 VALUES OF 的是一个集合，

它为 FORALL 计数器提供索引值，如实例 15-19 所示。

实例 15-19 使用 VALUES OF。

```
SQL> declare
  2         type type1 is table of number        index by pls_integer;
  3          type type2 is table of varchar2(20)     index by pls_integer;
  4          type type3 is table of pls_integer   index by pls_integer;
  5          num_type1         type1;
  6          text_type2          type2;
  7          count_type3        type3;
  8          v_count            number;
  9       begin
 10        for i in 1..6 loop
 11           num_type1(i)  := i;
 12           text_type2(i)  := 'type'||i;
 13           count_type3(i)  := i;
 14        dbms_output.put_line('count_type3('||i||')  : '||count_type3(i));
 15        end loop;
 16      execute immediate 'truncate table test';
 17      forall i in values of count_type3
 18           insert into test
 19           values ( num_type1(i)  , text_type2(i) );
 20          commit;
 21      select count(*)
 22          into v_count
 23          from test;
 24      dbms_output.put_line('v_count : '||v_count);
 25   end;
```

在上例中定义了一个集合类型 type3，该集合中的数据类型为 PLS_INTEGER，在第 13 行的 FOR 循环中为该参数填充集合元素。在第 17 行的 FORALL 中使用该集合提供循环计数器的索引值。下面是执行上述过程的结果。

```
26  /
count_type3(1)  : 1
count_type3(2)  : 2
count_type3(3)  : 3
count_type3(4)  : 4
count_type3(5)  : 5
count_type3(6)  : 6
v_count : 6

PL/SQL 过程已成功完成。
```

从执行结果可以看出，PLS_INTEGER 类型的集合中有 6 个元素，这 6 个元素可以为 FORALL 的使用提供循环计数器的索引值，本质上这些索引值不是唯一的，而且可以按照任意顺序列出。

15.3.3 使用 BULK COLLECT

使用 BULK COLLECT 选项，可以实现 SQL 语句的批量检索结果，从而将所有的结果一并发送到 PL/SQL 引擎。在游标一章中，我们使用游标实现了对查询结果的多条记录的处理，即通过游标对检索结果的集合一次处理一行记录，例如实例 15-20 使用游标检索 HR 模式下 EMPLOYEES 的员工信息。

实例 15-20　使用游标检索数据。

```
SQL> declare
  2    cursor emp_cur is
  3        select employee_id,first_name,last_name,salary
  4         from employees
  5          where salary>15000;
  6   begin
  7     for e_cur in emp_cur loop
  8        dbms_output.put_line(e_cur.employee_id||e_cur.first_name||
  9        e_cur.last_name||e_cur.salary);
 10     end loop;
 11  end;
 12  /
100StevenKing24000
101NeenaKochhar17000
102LexDe Haan17000

PL/SQL 过程已成功完成。
```

在上例中，我们使用游标获得了记录的集合，然后使用 FOR 循环依次获得数据记录，而使用 BULK COLLECT 可以实现同样的功能，但是需要先定义几个集合类型。下面通过 BULK COLLECT 来批量处理检索到的数据，如实例 15-21 所示。

实例 15-21　使用 BULK COLLECT 检索数据。

```
SQL> set serveroutput on
SQL> l
  1  declare
  2       type type1  is table of number(6);
  3       type type2 is table  of varchar2(20);
  4       type type3 is table of varchar2(25);
  5       type type4 is table of number(8,2);
  6       type1_tab  type1;
  7       type2_tab  type2;
  8       type3_tab  type3;
  9       type4_tab  type4;
 10   begin
 11           select employee_id,first_name,last_name,salary
 12            bulk collect into type1_tab,type2_tab,type3_tab,type4_tab
 13           from employees
 14         where salary>15000;
 15        for i in type1_tab.first .. type1_tab.last
 16        loop
 17           dbms_output.put_line(type1_tab(i));
 18           dbms_output.put_line(type2_tab(i));
 19           dbms_output.put_line(type3_tab(i));
 20           dbms_output.put_line(type4_tab(i));
 21       end loop;
 22*  end;
```

在上例中，创建了 4 个集合类型 TYPE1、TYPE2、TYPE3、TYPE4，因此在 BEGIN 部分的 SELECT 语句中使用 BULK INTO 将获得的多行记录批量填充到这 4 个集合中。此时集合中定义的元素数据类型必须和 SELECT 语句中选择的列的数据类型一致。在第 12 行使用了 COLLECT INTO 选项批量填充数据，第 15 行调用了集合 TYPE1 的属性 FIRST 和 LAST，用于提供 FOR 循环的索

引值。下面为执行该过程的结果。

```
SQL> /
100
Steven
King
24000
101
Neena
Kochhar
17000
102
Lex
De Haan
17000

PL/SQL 过程已成功完成。
```

这里有一个问题，当 SELECT 子句使用 BULK COLLECT 选项，不返回数据时不会报错，错误会在调用集合时触发，所以有必要判断 SELECT 子句是否返回数据从而填充集合，修改上例中的 SELECT 子句的谓词部分。

```
 11          select employee_id,first_name,last_name,salary
 12           bulk collect into type1_tab,type2_tab,type3_tab,type4_tab
 13          from employees
 14        where salary>30000;
```

再次运行匿名过程时，会报如下错误。

```
 15        for i in type1_tab.first .. type1_tab.last
 16        loop
 17           dbms_output.put_line(type1_tab(i));
 18           dbms_output.put_line(type2_tab(i));
 19           dbms_output.put_line(type3_tab(i));
 20           dbms_output.put_line(type4_tab(i));
 21       end loop;
 22   end;
 23  /
declare
*
第 1 行出现错误:
ORA-06502: PL/SQL: 数字或值错误
ORA-06512: 在 line 15
```

所以需要处理这个异常，方法是检查 SELECT 语句是否返回数据， 如果返回则在集合中存在数据；如果没有，则集合依然是空集合，所以可通过集合的属性来判断集合中是否存在数据。使用 BULK COLLECT 还有一个问题，就是如果返回数据过多应该如何处理呢？在 BULK COLLECT 选项中可使用 LIMIT 关键字限制每次读取的数据行数，如实例 15-22 所示。

实例 15-22　在 BULK COLLECT 中使用 LIMIT。

```
SQL> declare
  2       cursor emp_cur is
  3         select employee_id,first_name,last_name,salary
  4          from employees
  5            where salary>15000;
```

```
  6      type type1  is table of number(6);
  7      type type2 is table  of varchar2(20);
  8      type type3 is table of varchar2(25);
  9      type type4 is table of number(8,2);
 10      type1_tab  type1;
 11      type2_tab  type2;
 12      type3_tab  type3;
 13      type4_tab  type4;
 14      var_limit pls_integer :=20;
 15    begin
 16     open emp_cur;
 17      loop
 18           fetch emp_cur
 19           bulk collect into type1_tab,type2_tab,type3_tab,type4_tab
 20           limit var_limit;
 21           exit when type1_tab.count =0;
 22           for i in type1_tab.first .. type1_tab.last
 23         loop
 24
 25         dbms_output.put_line(type1_tab(i)||type2_tab(i)||type2_tab(i)
          ||type4_tab(i));
 26         end loop;
 27     end loop;
 28   close emp_cur;
 29  end;
 30  /
100StevenSteven24000
101NeenaNeena17000
102LexLex17000

PL/SQL 过程已成功完成。
```

在上例中定义一个 PLS_INTEGER 变量，并赋予初值，且定义了一个游标，使用 FETCH 填充集合，在第 19 行和第 20 行中使用 BULK COLLECT…LIMIT 限制了返回的数据量。

15.4　本章小结

本章介绍了 PL/SQL 中的 SQL 语句使用方法，这些 SQL 语句包括静态 SQL 语句和动态 SQL 语句，其中静态 SQL 语句是明确无误的，而动态 SQL 语句在运行时确定，这显然增加了程序的灵活性，更能适应实际的业务需求。

在静态 SQL 语句部分简单介绍了如何使用静态 SQL 语句来填充变量，以及如何通过 DML 语句实现过程中的部分功能操作。在动态 SQL 语句部分详细介绍了如何使用 EXECUTE IMMEDIATE 来执行动态 SQL 语句，这些 SQL 语句包括有效的 SQL 语句和 SQL 语句块，同时介绍了在动态 SQL 语句中如何使用 USING 和 RETURNING INTO 子句，并给出了执行动态 SQL 语句的注意事项，最后介绍了如何使用 FORALL 实现批处理。

第 16 章

PL/SQL调试

任何编程语言都会提供相应的程序调试方式或具体工具，从而跟踪程序的执行，完成调试任务。本章将介绍 Oracle 的调试工具，例如，通过常用的 DBMS_OUTPUT 包来打印缓存中的文本消息，通过 DBMS_UTILITY.FORMAT_CALL_ STACK 包来跟踪程序的执行位置等。

16.1　DBMS_OUTPUT包

这个工具包是 Oracle 内置的，使用该包的相关函数可将文本信息返回到启动 DBMS_OUTPUT 的客户端上。它是调试 PL/SQL 的最简单方法，正如在 Java 语言中使用 System.output.println()函数类似。在使用 DBMS_OUTPUT 打印文本消息的长度是有限制的，缓冲区有 1000 000 个字节，每行最多 255 个字符。DBMS_OUTPUT 将需要打印的文本消息缓存到一个 VARCHAR2(255)类型的数组中。

16.1.1　在 PL/SQL 调试中调用 DBMS_OUTPUT 包

如何在客户端使用 DBMS_OUTPUT 包，并打印文本消息呢？下面通过实例 16-1 进行演示，此时调用 DBMS_OUTPUT 包的过程 PUT_LINE。

实例 16-1　声明一个匿名过程来验证 DBMS_OUTPUT 包的过程 PUT_LINE。

```
SQL> declare
  2   i number := &num;
  3  begin
  4   dbms_output.put_line(i);
  5   dbms_output.put_line(i+1);
  6  end;
  7  /
输入 num 的值:  1000
原值    2:   i number := &num;
新值    2:   i number := 1000;

PL/SQL 过程已成功完成。
```

在上例中创建了一个匿名过程，首先声明一个整型变量 i，该变量的值需要在运行时绑定，在过程的主体部分调用包 DBMS_OUTPUT 的过程 PUT_LINE，首先显示该变量 i 的初始值，然后为变量 i 加 1，并打印数值。从上述分析可以知道，该过程会在启动了 DBMS_OUTPUT 包的客户端

打印两行数据；一个是 i 的原始值；另一个是加 1 后的值。但是从输出可以知道，上例中执行的匿名过程并没有打印我们需要的信息。这里需要注意，若要显示文本信息，需要在客户端启动 DBMS_OUTPUT 包，如实例 16-2 所示。

实例 16-2　执行匿名过程。

```
SQL> set serveroutput on;
SQL>declare
  2    i number := &num;
  3  begin
  4   dbms_output.put_line(i);
  5   dbms_output.put_line(i+1);
  6* end;
SQL> /
输入 num 的值:  1000
原值    2:   i number := &num;
新值    2:   i number := 1000;
1000
1001

PL/SQL 过程已成功完成。
```

上例在客户端启动了 DBMS_OUTPUT 包，执行了指令 set serveroutput on；再次执行匿名过程时在客户端打印了文本信息 i 的原始值 1000，以及 i+1 后的值 1001。

包 DBMS_OUTPUT 是由多个过程组成的，可以通过 DESC 指令查看该包的所有过程，以及过程涉及的参数（该实例需要 11g 版本的包结构），如实例 16-3 所示。

实例 16-3　包 DBMS_OUTPUT 包含的过程。

```
SQL> desc dbms_output;
PROCEDURE DISABLE
PROCEDURE ENABLE
参数名称                        类型                    输入/输出默认值?
------------------------------ ----------------------- ------ --------
 BUFFER_SIZE                    NUMBER(38)              IN     DEFAULT
PROCEDURE GET_LINE
参数名称                        类型                    输入/输出默认值?
------------------------------ ----------------------- ------ --------
 LINE                           VARCHAR2                OUT
 STATUS                         NUMBER(38)              OUT
PROCEDURE GET_LINES
参数名称                        类型                    输入/输出默认值?
------------------------------ ----------------------- ------ --------
 LINES                          TABLE OF VARCHAR2(32767) OUT
 NUMLINES                       NUMBER(38)              IN/OUT
PROCEDURE GET_LINES
参数名称                        类型                    输入/输出默认值?
------------------------------ ----------------------- ------ --------
 LINES                          DBMSOUTPUT_LINESARRAY   OUT
 NUMLINES                       NUMBER(38)              IN/OUT
PROCEDURE NEW_LINE
PROCEDURE PUT
参数名称                        类型                    输入/输出默认值?
------------------------------ ----------------------- ------ --------
 A                              VARCHAR2                IN
```

```
PROCEDURE PUT_LINE
参数名称                        类型                    输入/输出默认值?
------------------------------ ----------------------- ------ --------
 A                              VARCHAR2                IN
```

16.1.2　在 DBMS_OUTPUT 中应用 ENABLE 与 DISABLE 过程

通过 ENABLE 和 DISABLE 过程可以控制文本消息是否缓存并进行打印：调用 DBMS_OUTPUT.DISABLE 过程，可以禁止打印文本消息；使用 DBMS_OUTPUT.ENABLE 可以启动文本消息打印功能。通过下面的实例 16-4、实例 16-5 演示其使用。

实例 16-4　测试 DBMS_OUTPUT 的 ENABLE 与 DISABLE 过程。

```
SQL> set serveroutput on;
SQL> declare
  2    i number := &num;
  3  begin
  4   dbms_output.disable; --禁止打印文本消息
  5
  6   dbms_output.put_line(i);
  7   dbms_output.put_line(i+1);
  8
  9   dbms_output.enable;  --启动打印文本消息功能
 10
 11   dbms_output.put_line(i+3);
 12  end;
 13  /
输入 num 的值:  1000
原值    2:   i number := &num;
新值    2:   i number := 1000;
1003

PL/SQL 过程已成功完成。
```

在匿名过程的第 4 行，禁止打印文本消息，这样在重新启动打印文本消息之前的所有缓冲区消息都不会打印到客户端，所以第 6 行和第 7 行的打印消息都没有在客户端显示，接着在第 9 行启动打印文本消息功能，这样在禁止打印文本消息之前的消息都可以在客户端显示，所以在匿名过程的显示结果中只有第 11 行的消息，即 i+3 的结果是 1003。



DBMS_OUTPUT.DISABLE 的作用是清除任何缓冲区的消息，即使在该过程执行之前执行了 DBMS_OUTPUT.ENABLE 过程也是如此。

实例 16-5　测试 DBMS_OUTPUT.DISABLE 清除任何缓冲区。

```
SQL> declare
  2    i number := &num;
  3  begin
  4   dbms_output.disable;       --禁止打印文本消息
  5   dbms_output.put_line(i);
  6
  7   dbms_output.enable;        --启动打印文本消息功能
  8   dbms_output.put_line(i+1);
  9
```

```
 10   dbms_output.disable;        --再次禁止打印文本消息
 11   dbms_output.put_line(i+3);
 12  end;
 13  /
输入 num 的值:  1000
原值    2:   i number := &num;
新值    2:   i number := 1000;

PL/SQL 过程已成功完成。
```

在上例中，虽然在第 7 行启动了消息打印功能，但是第 8 行的信息依然无法显示，因为在第 10 行调用了 DBMS_OUTPUT.DISABLE 指令，清空了缓冲区，所以无法显示，而第 11 行的信息也无法打印。

16.2 DBMS_UTILITY包

该包的作用是返回当前调用栈的格式化文本字符，从而精确找出代码执行到的位置，反映了程序的调用关系。下面将创建三个过程，如实例 16-6、实例 16-7、实例 16-8 所示。

实例 16-6 创建过程 I。

```
SQL> create or replace
  2  procedure I as
  3  begin
  4    dbms_output.put_line(dbms_utility.format_call_stack);
  5  end I;
  6  /

过程已创建。
```

该过程调用 DBMS_UTILITY.FORMAT_CALL_STACK，并打印其格式化输出。

实例 16-7 创建过程 YOU，该过程调用过程 I。

```
SQL> create or replace
  2  procedure YOU as
  3   begin
  4     I;
  5   end YOU;
  6  /

过程已创建。
```

实例 16-8 创建过程 HE，该过程调用过程 YOU。

```
SQL> create or replace
  2  procedure HE as
  3  begin
  4    YOU;
  5  end HE;
  6  /

过程已创建。
```

下面分别按照顺序执行三个过程，如实例 16-9、实例 16-10、实例 16-11 所示。

实例 16-9　执行过程 I。

```
SQL> begin
  2  I;
  3  end;
  4  /
----- PL/SQL Call Stack -----
  object      line  object
  handle    number    name

2E1DB2B0         3  procedure SYS.I
2E2D570C         2  anonymous block

PL/SQL 过程已成功完成。
```

输出的打印结果从下向上顺序执行，该过程会执行一个匿名过程，然后调用过程 I。

实例 16-10　执行过程 YOU。

```
SQL> begin
  2  YOU;
  3  end;
  4  /
----- PL/SQL Call Stack -----
  object      line  object
  handle    number    name

2E1DB2B0         3  procedure SYS.I
28D9AE16         3  procedure SYS.YOU
33DCB200         2  anonymous block

PL/SQL 过程已成功完成。
```

输出的打印结果从下向上顺序执行，该过程会执行一个匿名过程，然后调用过程 YOU，最后调用过程 I。

实例 16-11　执行过程 HE。

```
SQL> begin
  2  HE;
  3  end;
  4  /
----- PL/SQL Call Stack -----
  object      line  object
  handle    number
name
2E1DB2B0         3  procedure   SYS.I
28D9AE16         3  procedure   SYS.YOU
28D9BB84         3  procedure   SYS.HE
33DD805C         2  anonymous  block
```

输出的打印结果从下向上顺序执行，该过程会执行一个匿名过程，然后调用过程 HE，过程 HE 调用过程 YOU，过程 YOU 又调用过程 I。

从三个过程的调用以及打印当前调用栈的信息可以看出 PL/SQL 代码的执行顺序，以及代码执行到了哪个具体位置。

16.3 自治事务

使用自治事务的好处是可以将需要的调试信息都记录在数据库表中，即使发生事务回滚依然可以 记录这些异常信息，可供用户查询。下面通过实例具体说明如何使用自治事务。

首先创建两个表：auto_log_t、auto_test_t，其中 auto_log_t 用于记录调试信息，即发生异常时记录的错误消息；auto_test_t 用于 PL/SQL 代码中实际使用的表，即用户需要的表。创建测试表，如实例 16-12 所示。

实例 16-12 创建测试表。

```
SQL> create table auto_log_t (
  2          auto_date   date,
  3          auto_mesg   clob);

表已创建。

SQL> create table auto_test_t
  2      ( auto_val  varchar2(20));

表已创建。
```

接下来需要创建一个包，该包包含一个过程，在编写 PL/SQL 代码时，可以使用该过程来记录错误消息，如实例 16-13、实例 16-14 所示。

实例 16-13 创建包 auto_log。

```
SQL> create or replace
  2   package auto_log as
  3    procedure record_errors(err_mesg varchar2);
  4   end auto_log;
  5  /

程序包已创建。
```

实例 16-14 创建包体。

```
SQL> create or replace
  2   package body auto_log as
  3    procedure record_errors(err_mesg varchar2) is
  4      pragma  autonomous_transaction;
  5      begin
  6        insert into auto_log_t
  7          values(sysdate,err_mesg);
  8        commit;
  9      end record_errors;
 10  end auto_log;
 11  /

程序包体已创建。
```

在编写 PL/SQL 代码时调用 record_errors 来记录这些异常信息。如果没有发生异常，则在表 auto_log_t 中记录成功标识，否则再记录异常信息。创建过程 test_log，如实例 16-15 所示。

实例 16-15　创建过程 test_log。

```
SQL> create or replace
  2  procedure test_log (v_mesg  varchar2) as
  3  begin
  4    auto_log.record_errors('Starting......');
  5    insert into auto_test_t values(v_mesg);
  6    auto_log.record_errors('Ending......');
  7  commit;          --没有异常则提交事务
  8  exception
  9   when others then  --异常发生时记录错误信息
 10    auto_log.record_errors('当插入数据'||v_mesg||'时发生异常');
 11     auto_log.record_errors(SQLCODE||'and'||SQLERRM);
 12    rollback;       --异常发生时回滚事务，此时错误消息已经记录在 auto_log_t 表中了。
 13  end test_log;
 14  /

过程已创建。
```

目前已创建记录异常信息的过程 record_errors 以及 test_log。下面尝试使用过程 test_log 测试使用自治事务记录异常的功能，如实例 16-16 所示。

实例 16-16　调用过程 test_log。

```
SQL> exec test_log('hello PL/SQL');

PL/SQL 过程已成功完成。
```

下面查询表 auto_test_t 是否记录了数据，如实例 16-17 所示。

实例 16-17　查看表 auto_test_t。

```
SQL> select * from auto_Test_t;

AUTO_VAL
--------------------
hello PL/SQL
```

此时事务得以提交并且没有异常发生，否则在该表中将看不到数据。下面查询表 auto_log_t 中记录了哪些信息，如实例 16-18 所示。

实例 16-18　查看表 auto_log_t。

```
SQL>select to_char(auto_date,'yyyy-mm-dd hh24:mi:ss'),auto_mesg from auto_ log_t;

TO_CHAR(AUTO_DATE,' AUTO_MESG')
------------------- --------------------
2012-09-02 07:45:41 Starting......
2012-09-02 07:45:41 Ending......
```

从输出可以看出，此时没有异常发生。下面构建一个异常语句，查看是否记录了这个错误消息，如实例 16-19 所示。

实例 16-19　调用过程 test_log。

```
SQL>exec test_log('Oracle Database 11g Enterprise Edition Release 11.2.0.1.0');
```

```
PL/SQL 过程已成功完成。
```

此时，我们插入的数据超过了参数定义的长度，将触发异常。下面查看表 auto_log_t 是否记录了异常，如实例 16-20 所示。

实例 16-20　查看表 auto_log_t。

```
SQL>select to_char(auto_date,'yyyy-mm-dd hh24:mi:ss'),auto_mesg from auto_log_t;

TO_CHAR(AUTO_DATE,'AUTO_MESG')
--------------------------------------------------------------
2012-09-02 07:45:41 Starting......
2012-09-02 07:45:41 Ending......
2012-09-02 07:50:33 Starting......
2012-09-02 07:50:33 当插入数据 Oracle Database 11g Enterprise Edition Release
 11.2.0.1.0 时发生异常
2012-09-02 07:50:33 -12899andORA-12899: 列 "SYS"."AUTO_TEST_T"."AUTO_VAL" 的
值太大 (实际值: 57)
```

从输出可以知道，此时发生了异常，并且该异常信息被过程 test_log 的异常语句捕获，记录在表 auto_log_t 表中，虽然事务回滚，插入的数据没有成功，但是异常错误信息却成功地记录下来，在 PL/SQL 代码调试时，就可以从表 auto_log_t 获得过程 auto_test 调用中发生的所有异常。

16.4　UTL_FILE包

UTL_FILE 是 Oracle 提供的一个从 PL/SQL 过程向文件系统写入数据的包，通过该包提供的函数，可以打开操作系统的文件，并调用相关函数向文件写入数据，这些数据可以通过 PL/SQL 过程从 Oracle 中读取。在调试 PL/SQL 程序时，也可以将调试信息写入文件，从而可以通过读取并分析文件来分析程序错误。下面是 UTL_FILE 包的相关函数的语法以及注意事项，最后通过一个实例演示如何具体使用该包。

使用 UTL_FILE 包需要一个前提，即创建 DIRECTORY 目录，并赋予用户相应的权限，如实例 16-21、实例 16-22 所示。

实例 16-21　创建目录对象。

```
SQL> create or replace directory file_dir as
  2  'c:/test';

目录已创建。
```

实例 16-22　赋予用户 lin 读写该目录的权限。

```
SQL> grant read ,write on directory file_dir to lin;

授权成功。
```

1. 打开文件

FOPEN 会打开指定文件并返回一个文件句柄用于操作文件，其语法如下所示。

```
  FUNCTION UTL_FILE.FOPEN (
               location     IN VARCHAR2,
```

```
            filename     IN VARCHAR2,
            open_mode    IN VARCHAR2,
            max_linesize IN BINARY_INTEGER)
 RETURN file_type;
```

对上述参数的说明如下。

- location：文件所在的目录。可以是使用 CREATE DIRECTORY 创建的目录对象，也可以是绝对路径的真实目录。
- filename：文件名称，包括文件类型（如.TXT）。
- open_mode：打开文件的模式，有 3 种文件打开模式。
    - R 只读模式：一般配合 UTL_FILE 的 GET_LINE 来读文件。
    - W 写（替换）模式：文件的所有行会被删除。PUT、PUT_LINE、NEW_LINE、PUTF 和 FFLUSH 都可使用。
    - A 写（附加）模式：原文件的所有行会被保留，在最末尾附加新行。PUT、PUT_LINE、NEW_LINE、PUTF 和 FFLUSH 都可使用。

在使用打开文件函数时需要注意以下几个问题。

- 文件路径和文件名合起来必须表示操作系统中一个合法的文件。
- 文件路径必须存在并可访问，FOPEN 并不会新建一个文件夹。
- 如果想打开文件进行读操作，文件必须存在；如果想打开文件进行写操作，文件不存在时会新建一个文件。
- 如果想打开文件进行附加操作，文件必须存在。文件不存在时，会抛出 INVALID_OPERATION 异常。

2. 写入文件

UTL_FILE.PUT_LINE 函数用于实现向文件写入 PL/SQL 过程读取的数据。其语法格式如下所示。

```
PROCEDURE UTL_FILE.PUT_LINE
    (file IN UTL_FILE.FILE_TYPE,
    buffer IN VARCHAR2);
file
```

由 FOPEN 返回的文件句柄 buffer 包含要写入文件的数据缓存，在调用 UTL_FILE.PUT_LINE 前，必须先打开文件，否则 UTL_FILE.PUT_LINE 会产生以下异常：

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.INVALID_OPERATION
- UTL_FILE.WRITE_ERROR

3. 关闭文件

UTL_FILE.FCLOSE 用于关闭文件，其语法格式如下所示。

```
PROCEDURE UTL_FILE.FCLOSE (file IN OUT FILE_TYPE);
file
```

注意 file 是一个 IN OUT 参数，因为在关闭文件后会设置为 NULL，当试图关闭文件时，若有缓存数据未写入文件，将抛出 WRITE_ERROR 异常，UTL_FILE.FCLOSE 会产生以下异常：

- UTL_FILE.INVALID_FILEHANDLE
- UTL_FILE.WRITE_ERROR

在介绍了 UTL_FILE 的三个函数之后，下面通过实例 16-23 学习如何使用它们将 Oracle 中的信息写入操作系统的文件。

实例 16-23 使用 UTL_FILE 包的函数。

```
SQL> declare
  2    fileID UTL_FILE.FILE_TYPE;  --定义文件句柄
  3  BEGIN
  4    fileID := UTL_FILE.FOPEN ('FILE_DIR', 'dept.TXT', 'W'); --获得文件句柄
  5
  6    FOR deptrec IN (SELECT * FROM dept)
  7    LOOP
  8      UTL_FILE.PUT_LINE                         --通过获得文件句柄向文件写数据
  9         (FILEID,TO_CHAR (deptrec.dname) || ',' ||
 10          deptrec.loc || ',' ||
 11          TO_CHAR (deptrec.deptno));
 12    END LOOP;
 13    UTL_FILE.FCLOSE (fileID);                    --关闭文件。
 14  END;
 15  /

PL/SQL 过程已成功完成。
```

在上例的目录对象 file_dir 中自动创建了一个文件 dept.txt，因为这是 W（写入）模式，所以如果没有该文件，会自动创建操作系统文件。通过循环读取游标获得语句 SELECT * FROM dept 读取的数据，然后通过 PUT_FILE 函数写入文件 dept.txt。写入的数据内容如图 16-1 所示。

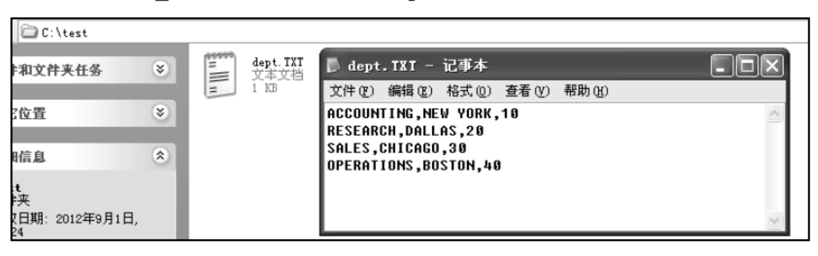


图 16-1 PUT_FILE 函数写入文件 dept.txt

上图的内容首先显示了在目录 c:/test 下创建了文件 dept.txt，通过打开该文件，发现其中确实写入了我们读取的表 dept 中的数据。

下面再具体介绍一个实例 16-24，用于说明使用 UTL_FILE 包如何将异常信息写入操作系统文件。该实例需要利用自治事务的实例，但要改写包 auto_log 的过程 record_errors。

实例 16-24 修改包 auto_log。

```
SQL> create or replace
  2  package body auto_log as
  3  procedure record_errors(err_mesg varchar2) is
```

```
  4   file_id utl_file.file_type;
  5  begin
  6   file_id := utl_file.fopen('FILE_DIR','errors.txt','a');
  7   utl_file.put_line(file_id,to_char(sysdate,'yyyy-mm-dd hh24:mi:ss')||''
      ||err_mesg);
  8   utl_file.fclose(file_id);
  9   exception
 10    when others then
 11    null;
 12    end record_errors;
 13    end auto_log;
 14   /

程序包体已创建。
```

在上例中，我们修改了过程 record_errors 的内容，其程序主体部分使用 UTL_FILE 包通过对操作系统文件的操作来记录过程 auto_log 的异常信息，记录在文件 errors.txt 中。下面做一个测试，如实例 16-25 所示。

实例 16-25　测试。

```
SQL> exec test_log('Oracle Database 11g Enterprise Edition');

PL/SQL 过程已成功完成。
```

下面打开目录对象 file_dir 下的文件 errors.txt，查看其是否记录了异常信息，如图 16-2 所示。



图 16-2　查看信息

从图中可以看出，文件记录了异常发生的时间，以及对应的所有错误信息，在 PL/SQL 代码调试时，就可以使用该文件获取调试信息。

16.5　本章小结

本章介绍了 PL/SQL 程序调试的常用工具，其本质是 Oracle 提供的各种包以及函数。这些工具包括 DBMS_OUTPUT 包、DBMS_UTILITY 包、UTL_FILE 文件操作包以及使用自治事务。熟悉这些工具的特点、使用方法以及各自适合的环境后，在编写 PL/SQL 程序时就可以选择适合自己的代码调试方式。

第 17 章

常用工具包

本章介绍的常用工具包括嵌入 Oracle 数据库软件的工具包和用户自己编写的工具包，只要在合适的时机以及利用适当的方法就可以有效地提高工作效率，这些包包括 Oracle 提供的包、处理警告日志文件的包、数据库维护包以及监控数据库历史的包。

17.1 Oracle提供的包

Oracle 提供的包集成在 Oracle 数据库软件中，只要安装了 Oracle 的 RDBMS 就可以使用这些包。本节将介绍调度管理包、审计包、解析 SQL 执行计划包以及 DBMS_HPROF 包的使用。

17.1.1 调度管理包

Oracle 为了更加自动化地管理数据库的诸多任务，可使用调度包完成任务的调度，这些任务可以是 PL/SQL 代码，也可以是过程，调度包完成任务与时间窗口的结合，使得在某个特定的时间点或者时间段执行某项任务，如定期重建索引等。下面可以通过 DESC 指令获得包的结构信息，如下是包 DBMS_JOB 包含的过程。

```
FUNCTION BACKGROUND_PROCESS RETURNS BOOLEAN
PROCEDURE BROKEN
PROCEDURE CHANGE
PROCEDURE INSTANCE
PROCEDURE INTERVAL
PROCEDURE ISUBMIT
FUNCTION IS_JOBQ RETURNS BOOLEAN
PROCEDURE NEXT_DATE
ROCEDURE REMOVE
PROCEDURE RUN
PROCEDURE SUBMIT
PROCEDURE USER_EXPORT
PROCEDURE USER_EXPORT
PROCEDURE WHAT
```

DBMS_JOB 包是向用户提供的一个 PL/SQL 包，通过它向作业队列提供一个作业，并在指定的时间或者指定的时间段运行，同时用户可以通过 INTERVAL 过程改变作业执行的频繁程度。下面解释包的主要过程，然后通过具体的实例说明如何使用该包来完成作业调度。

1. SUBMIT

这是一个过程，用来向作业队列中提交 PL/SQL 过程。该过程的参数如下所示。

```
PROCEDURE SUBMIT
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 JOB                            BINARY_INTEGER          OUT
 WHAT                           VARCHAR2                IN
 NEXT_DATE                      DATE                    IN     DEFAULT
 INTERVAL                       VARCHAR2                IN     DEFAULT
 NO_PARSE                       BOOLEAN                 IN     DEFAULT
 INSTANCE                       BINARY_INTEGER          IN     DEFAULT
 FORCE                          BOOLEAN                 IN     DEFAULT
```

在提交作业之前，需要清楚地理解 DBMS_JOB.SUBMIT 的过程参数含义，它包括 5 个参数，其中 4 个 IN 参数，1 个 OUT 参数（提交作业的编号）。

- JOB：标识队列中某个作业的编号。
- WHAT：部分 PL/SQL 过程和参数。
- NEXT_DATE：下次作业执行的时间。
- INTERVAL：作业下一次运行的时间。
- NO_PARSE：是 BOOLEAN 指示器，决定在提交作业时是否运行该作业，默认为不运行。

2. REMOVE

REMOVE 用于从作业队列中删除已经提交的 PL/SQL 过程，该过程只有一个参数：JOB 作业名称，该名称使用一个数值表示，数据类型为 BINARY_INTEGER。

3. CHANGE

CHANGE 用于用于修改提交作业的参数，此时需要把握该过程的参数含义，其参数如下所示。这些可修改的作业参数包括作业名称、描述、运行时间间隔、下次运行时间等。在运用该过程时我们再讨论这些参数的含义，这里不再赘述。

```
Argument Name                  Type              In/Out Default?
 ---------------------------- ----------------- --------------------
 JOB                            BINARY_INTEGER    IN
 WHAT                           VARCHAR2          IN
 NEXT_DATE                      DATE              IN
 INTERVAL                       VARCHAR2          IN
 INSTANCE                       BINARY_INTEGER    IN     DEFAULT
 FORCE                          BOOLEAN           IN     DEFAULT
```

4. BROKEN

BROKEN 用于禁止队列中的某个作业。

5. INTERVAL

INTERVAL 用于修改已经提交的作业的执行时间间隔。

6. NEXT_DATE

NEXT_DATE 用于修改已经提交的作业的下次运行时间。

7. RUN

RUN 用于运行作业队列中的某个作业，直接执行不必考虑是否位于调度时间窗口内。

作业队列由后台进程控制，进程数量由参数 JOB_QUEUE_PROCESSES 决定，在有多个调度作业时，必须合理设置该参数的值，以免因为进程数量不够而影响数据库性能。参数 JOB_QUEUE_PROCESSES 可以在初始化参数中设置，也可以动态修改，如实例 17-1 所示。

实例 17-1　修改参数 JOB_QUEUE_PROCESSES。

```
SQL> alter system set job_queue_processes=200 scope=both;

System altered.

SQL> show parameter job_queue_processes;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
job_queue_processes                  integer     200
```

下面我们创建一个 PL/SQL 过程。该过程的作用是对模式 HR 的表 EMPLOYEES 的索引进行重建，如实例 17-2 所示。

实例 17-2　创建一个过程 rebuild_index_prod。

```
SQL> create or replace procedure rebuild_index_prod
  2   is
  3     idx_name varchar2(20);
  4     cursor cur_idx_rebuild
  5      is
  6         select index_name from user_indexes where table_name='EMPLOYEES';
  7      begin
  8         open cur_idx_rebuild;
  9         loop
 10            fetch cur_idx_rebuild into idx_name;
 11             execute immediate 'alter index '||idx_name ||' rebuild';
 12        dbms_output.put_line('idx_name is:'||idx_name||'has been rebuilt');
 13            exit when cur_idx_rebuild%notfound;
 14         end loop;
 15             dbms_output.put_line('end index rebuilding!');
 16* end;

Procedure created.
```

在创建完过程 rebuild_idx_prod 后，必须保证该过程可以正确执行，因为调度包的 SUBMIT 过程不检验过程是否正确，这点一定要注意，下面验证该过程是否创建完成，如实例 17-3 所示。

实例 17-3　验证过程是否创建。

```
SQL> select object_name,status,object_type from user_objects where object_name
 ='REBUILD_INDEX_PROD'

OBJECT_NAME                        STATUS  OBJECT_TYPE
---------------------------------- ------- -------------------
REBUILD_INDEX_PROD                  VALID   PROCEDURE
```

下面执行该过程，以确定它是否有效，如实例 17-4 所示。

实例 17-4　运行过程 rebuild_index_prod。

```
SQL> exec rebuild_index_prod
idx_name is :EMP_NAME_IXhas been rebuilt
idx_name is :EMP_MANAGER_IXhas been rebuilt
idx_name is :EMP_JOB_IXhas been rebuilt
idx_name is :EMP_DEPARTMENT_IXhas been rebuilt
idx_name is :EMP_EMP_ID_PKhas been rebuilt
idx_name is :EMP_EMAIL_UKhas been rebuilt
idx_name is :EMP_EMAIL_UKhas been rebuilt
end index rebuilding!

PL/SQL procedure successfully completed.
```

下面提交该过程，使得该过程每隔几天运行一次，如实例 17-5 所示。

实例 17-5　通过 DBMS_JOB 包提交过程。

```
SQL>declare
  2     var_job_num  number;
  3     begin
  4       dbms_job.submit(
  5                 job        => var_job_num,
  6                 what       => 'REBUILD_INDEX_PROD
  7                              (''/u01/app'', ''job.log'');',
  8                 next_date   =>sysdate,
  9                 interval    =>'sysdate + 1/2' );
 10        commit;
 11   dbms_output.put_line(var_job_num);
 12*  end;
```

以上代码提交了一个 JOB，可以通过包 DBMS_JOB 的 RUN 过程运行该 JOB，也可以使用 EM 或者 GC 运行该 JOB，这样就可以实现自动的任务调度功能，从而提高了数据库的维护效率。

17.1.2　审计包

审计是 Oracle 数据库在数据库层面的安全措施，可以有效记录用户对数据库的各种操作行为，如数据库登录、退出、对表数据的增删查改等，都可以通过审计功能实现，Oracle 提供了审计包以方便 DBA 实现审计操作。下面使用 DBMS_FGA 包来完成审计功能。DBMS_FGA 包的结构如实例 17-6 所示。

实例 17-6　查看包 DBMS_FGA 的结构。

```
SQL> desc dbms_fga;
PROCEDURE ADD_POLICY
参数名称                        类型                     输入/输出默认值?
------------------------------ ----------------------- ------ --------
 OBJECT_SCHEMA                  VARCHAR2                IN     DEFAULT
 OBJECT_NAME                    VARCHAR2                IN
 POLICY_NAME                    VARCHAR2                IN
 AUDIT_CONDITION                VARCHAR2                IN     DEFAULT
 AUDIT_COLUMN                   VARCHAR2                IN     DEFAULT
 HANDLER_SCHEMA                 VARCHAR2                IN     DEFAULT
 HANDLER_MODULE                 VARCHAR2                IN     DEFAULT
 ENABLE                         BOOLEAN                 IN     DEFAULT
```

```
 STATEMENT_TYPES              VARCHAR2               IN     DEFAULT
 AUDIT_TRAIL                  BINARY_INTEGER         IN     DEFAULT
 AUDIT_COLUMN_OPTS            BINARY_INTEGER         IN     DEFAULT
PROCEDURE DISABLE_POLICY
参数名称                         类型                   输入/输出默认值?
------------------------------ ----------------------- ------ --------
 OBJECT_SCHEMA                VARCHAR2               IN     DEFAULT
 OBJECT_NAME                  VARCHAR2               IN
 POLICY_NAME                  VARCHAR2               IN
PROCEDURE DROP_POLICY
参数名称                         类型                   输入/输出默认值?
------------------------------ ----------------------- ------ --------
 OBJECT_SCHEMA                VARCHAR2               IN     DEFAULT
 OBJECT_NAME                  VARCHAR2               IN
 POLICY_NAME                  VARCHAR2               IN
 PROCEDURE ENABLE_POLICY
参数名称                         类型                   输入/输出默认值?
------------------------------ ----------------------- ------ --------
 OBJECT_SCHEMA                VARCHAR2               IN     DEFAULT
 OBJECT_NAME                  VARCHAR2               IN
 POLICY_NAME                  VARCHAR2               IN
 ENABLE                       BOOLEAN                IN     DEFAULT
```

各个过程的功能如下。

- ADD_POLICY：增加审计策略以及审计对象。
- DISABLE_POLICY：禁止审计。
- DROP_POLICY：删除审计策略。
- ENABLE_POLICY：启动审计。

FGA（细粒度审计）从 Oracle 9i 开始引入，FGA 不但对行和列进行精细化审计，而且还记录触发审计的语句。通过调用 DBMS_FGA 包来实现 FGA，将需要审计的数据内容作为一个策略，在数据库中定义。审计信息记录在数据字典表 fga_log$中，通过查询视图 dba_fga_audit_trail 可以获得 FGA 的审计数据。

在 HR 模式中有两个表对象：EMPLOYEES 和 DEPARTMENTS，我们希望当用户查询表 EMPLOYEES 中的 FINANCE（财务）部门的员工信息时进行审计，如实例 17-7 所示。

实例 17-7　查询财务部门的员工信息。

```
SQL> select department_id from departments where department_name='Finance';

DEPARTMENT_ID
-------------
          100
```

也就是当其他客户端查询表 EMPLOYEES 中的 DEPARTMENT_ID=100 的员工信息时需要审计，记录用户操作。下面开始创建这个审计，如实例 17-8 所示。

实例 17-8　使用包 DBMS_FGA 创建审计。

```
SQL> begin
dbms_fga.add_policy(
object_schema=>'HR',
object_name=>'EMPLOYEES',
```

```
policy_name=>'audit_emp',
audit_condition=>'department_id=100',
enable=>true,
statement_types=>'SELECT,UPDATE'
);
END;
/

PL/SQL procedure successfully completed.
```

此时增加了一个细粒度审计策略，策略名为 audit_emp，下面解释函数 add_policy 中参数的含义。

- object_schema：指定审计的模式名。
- object_name：审计对象名称。
- policy_name：审计策略名。
- audit_condition：审计的触发条件（谓词部分）。
- enable：是否启动审计。
- statement_types：触发审计的动作。

这个审计策略的含义是当使用 SELECT、UPDATE 操作表，且谓词为'department_id=100'，即当操作财务部门的员工信息时审计。审计记录可通过 dba_audit_trail 查询，如实例 17-9 所示。

实例 17-9　执行查询触发审计。

```
SQL> connect hr/oracle
Connected.
SQL> select count(*) from employees where department_id=100;

  COUNT(*)
----------
         6
```

然后，通过数据字典 dba_fga_audit_trail 查看审计记录内容，如实例 17-10 所示。

实例 17-10　查看审计记录。

```
SQL> connect sys/oracle as sysdba
Connected.
SQL>select object_schema,object_name,sql_text,db_user from dba_fga_audit_ trail;

OBJECT_SCHEMA OBJECT_NAME  SQL_TEXT                                        DB_USER
--------------------------------------------------------------
HR       EMPLOYEES select count(*) from employees where department_id=100  HR
```

还可以在列级别上创建审计。例如在表 EMPLOYEES 上的列 SALARY 进行 SELECT 操作时进行审计，如实例 17-11 所示。

实例 17-11　在列级别上创建审计。

```
SQL> begin
dbms_fga.add_policy(
object_schema=>'HR',
object_name=>'EMPLOYEES',
```

```
policy_name=>'audit_emp_salary',
audit_column=>'SALARY',
enable=>true,
statement_types=>'SELECT'
);
END;
/
PL/SQL procedure successfully completed.
```

这里的关键是 audit_column，用于表明审计的列为 SALARY，下面做一次测试，对表 EMPLOYEES 执行一次查询，如实例 17-12 所示。

实例 17-12　执行一次查询。

```
SQL> select department_id,salary from employees where department_id=10;

DEPARTMENT_ID     SALARY
------------- ----------
           10       4400
```

查询审计记录，如实例 17-13 所示。

实例 17-13　查询审计内容

```
SQL>select object_schema,object_name,sql_text,db_user from dba_fga_audit_ trail;

OBJECT_SCHEMA OBJECT_NAME  SQL_TEXT                                                      DB_USER
--------------------------------------------------------------------------------------
HR EMPLOYEES select department_id,salary from employees where department_ id=10HR
```

也可以创建将行和列结合起来的审计，这样只有当两个条件都满足时才会触发审计，如实例 17-14 所示。

实例 17-14　创建行和列结合的审计。

```
SQL> connect sys/oracle as sysdba
Connected.
SQL> begin
dbms_fga.add_policy(
object_schema=>'HR',
object_name=>'EMPLOYEES',
policy_name=>'audit_emp_salary_depart',
audit_condition=>'department_id=10',
audit_column=>'SALARY',
enable=>true,
statement_types=>'SELECT'
);
END;
/
PL/SQL procedure successfully completed.
```

此时，只有查询语句中存在 SALARY 列，并且查询谓词为 department_id=10 才会触发这个审计行为。

当不需要一个审计时，应将其删除，这样可以节约审计数据的存储空间，减少系统性能影响。删除一个审计策略，如实例 17-15 所示。

实例 17-15　使用 DBMS_FGA 包删除审计策略。

```
SQL> begin
dbms_fga.drop_policy(
object_schema=>'HR',
object_name=>'EMPLOYEES',
policy_name=>'audit_emp_salary_depart');
end;
/
PL/SQL procedure successfully completed.
```

17.1.3　解析 SQL 执行计划包

数据库的核心就是执行 SQL 语句，而对于关系数据库而言，需要分析 Oracle 引擎执行 SQL 语句的过程以确定该 SQL 语句是否高效地执行。显然我们需要一个工具来分析 SQL 语句的执行过程，从而更加友好地显示 SQL 语句的执行计划。

Oracle 从 9.2 版本起就引入了 DBMS_XPLAN 包，用于显示 SQL 语句的执行计划，帮助 DBA 或程序员更好地了解 SQL 的执行计划。从本质上讲 DMBS_XPLAN 包是读取两个视图中的数据：V$SQL 和 V$SQL_PLAN，使用这些信息来填充表 PLAN_TABLE。如果需要该表可以使用 Oracle Home 目录下的脚本来创建，在 Oracle 11g 中该目录为$ORACLE_HOME/rdbms/admin，文件名为 utlxplan.sql。

下面是表 PLAN_TABLE 的结构。

```
SQL> desc plan_table;
 Name                                      Null?    Type
 ---------- --------- --------------------------------------------
 STATEMENT_ID                                       VARCHAR2(30)
 PLAN_ID                                            NUMBER
 TIMESTAMP                                      DATE
 REMARKS                                            VARCHAR2(4000)
 OPERATION                                          VARCHAR2(30)
 OPTIONS                                            VARCHAR2(255)
 OBJECT_NODE                                      VARCHAR2(128)
 OBJECT_OWNER                                    VARCHAR2(30)
 OBJECT_NAME                                      VARCHAR2(30)
 OBJECT_ALIAS                                      VARCHAR2(65)
 OBJECT_INSTANCE                                  NUMBER(38)
 OBJECT_TYPE                                       VARCHAR2(30)
 OPTIMIZER                                       VARCHAR2(255)
 SEARCH_COLUMNS                                 NUMBER
 ID                                                NUMBER(38)
 PARENT_ID                                        NUMBER(38)
 DEPTH                                             NUMBER(38)
 POSITION                                          NUMBER(38)
 COST                                             NUMBER(38)
 CARDINALITY                                       NUMBER(38)
 BYTES                                             NUMBER(38)
 OTHER_TAG                                       VARCHAR2(255)
 PARTITION_START                                  VARCHAR2(255)
 PARTITION_STOP                                    VARCHAR2(255)
 PARTITION_ID                                       NUMBER(38)
 OTHER                                             LONG
 OTHER_XML                                       CLOB
```

```
DISTRIBUTION                                      VARCHAR2(30)
CPU_COST                                          NUMBER(38)
IO_COST                                           NUMBER(38)
TEMP_SPACE                                        NUMBER(38)
ACCESS_PREDICATES                                 VARCHAR2(4000)
FILTER_PREDICATES                                 VARCHAR2(4000)
PROJECTION                                        VARCHAR2(4000)
TIME                                              NUMBER(38)
QBLOCK_NAME                                       VARCHAR2(30)
```

我们在需要使用包 DBMS_XPLAN 时，必须创建该表，最好再创建一个同义词，其名称与 PLAN_TABLE 相同，如此设置后任何模式都可以访问自己的执行计划。创建表 PLAN_TABLE 的代码如实例 17-16 所示。

实例 17-16 生成表 PLAN_TABLE。

```
[oracle@localhost ~]$ sqlplus sys/oracle as sysdba

SQL*Plus: Release 11.2.0.1.0 Production on Tue Sep 25 08:47:28 2012

Copyright (c) 1982, 2009, Oracle.  All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> @$ORACLE_HOME/rdbms/admin/utlxplan.sql
Table created.

SQL> grant all on sys.plan_table to public;
Grant succeeded.

SQL> SQL> create public synonym plan_table for sys.plan_table;
Synonym created.
```

下面执行一个 SQL 语句，并将该语句的执行计划存储在表 PLAN_TABLE 中，然后解释该执行计划，如实例 17-17 所示。

实例 17-17 分析 SQL 语句并存储执行计划。

```
SQL>explain plan for
  2  SELECT  e1.first_name, e1.last_name, j.job_id, sum(e2.salary) total_sal
  3    FROM employees e1, employees e2, job_history j
  4    WHERE e1.employee_id = e2.manager_id
  5      AND e1.employee_id = j.employee_id
  6      AND e1.hire_date = j.start_date
  7    GROUP BY e1.first_name, e1.last_name, j.job_id
  8*   ORDER BY total_sal

Explained.
```

在该 SQL 语句执行完毕之后，Oracle 将执行计划存储在表 PLAN_TABLE 中，但是如果要直接查询该表信息，则需要设置选择的属性。下面使用包 DBMS_XPLAN 的 DISPLAY 函数来显示该 SQL 语句的执行计划，执行计划如实例 17-18 所示。

实例 17-18　查询存储在 PLAN_TABLE 表中的执行计划。

```
  1* select * from table(dbms_xplan.display)
SQL> /

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
---------
Plan hash value: 4189704726

-------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name                      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                           |   106 |  5936 |    10  (30)| 00:00:01 |
|   1 |  SORT ORDER BY                 |                           |   106 |  5936 |    10  (30)| 00:00:01 |
|   2 |   HASH GROUP BY                |                           |   106 |  5936 |    10  (30)| 00:00:01 |
|   3 |    NESTED LOOPS                |                           |       |       |            |          |
|   4 |     NESTED LOOPS               |                           |   106 |  5936 |     8  (13)| 00:00:01 |
|*  5 |      HASH JOIN                 |                           |   106 |  3710 |     7  (15)| 00:00:01 |

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
---------
|   6 |       TABLE ACCESS FULL        | EMPLOYEES                 |   107 |  2889 |     3   (0)| 00:00:01 |
|   7 |       TABLE ACCESS FULL        | EMPLOYEES                 |   107 |   856 |     3   (0)| 00:00:01 |
|*  8 |      INDEX UNIQUE SCAN         | JHIST_EMP_ID_ST_DATE_PK |     1 |       |     0   (0)| 00:00:01 |
|   9 |     TABLE ACCESS BY INDEX ROWID| JOB_HISTORY               |     1 |    21 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("E1"."EMPLOYEE_ID"="E2"."MANAGER_ID")
   8 - access("E1"."EMPLOYEE_ID"="J"."EMPLOYEE_ID" AND "E1"."HIRE_DATE"="J"."START_DATE")
```

如果需要读懂上述的执行计划，需要读者具备查询优化的具体技术才行，这里不再做过多解释，有兴趣的读者可以参阅 Oracle 的官方文档 Performance Tuning Guide。

除了上述方式外，我们当然可以通过表 PLAN_TABLE 直接查询，但是需要进行一些处理，如实例 17-19 所示。

实例 17-19　通过表 PLAN_TABLE 查询执行计划。

```
SQL>select rtrim (lpad (' ',2*level) ||
  2             rtrim (operation)    ||' '||
  3             rtrim (options)      ||' '||
  4             object_name      ||' '||
  5             partition_start      ||' '||
  6             to_char(partition_id)
  7             ) PLAN_TABLE_OUTPUT
  8        from plan_table
  9       connect by prior id = parent_id
 10*      start with id = 0

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------
  SELECT STATEMENT
    SORT ORDER BY
      HASH GROUP BY
        NESTED LOOPS
          NESTED LOOPS
            HASH JOIN
              TABLE ACCESS FULL EMPLOYEES
              TABLE ACCESS FULL EMPLOYEES
```

```
            INDEX UNIQUE SCAN JHIST_EMP_ID_ST_DATE_PK
          TABLE ACCESS BY INDEX ROWID JOB_HISTORY

10 rows selected.
```

显然，直接通过表 PLAN_TABLE 获得的执行计划结果远没有使用包 XDBMS_PLAN 获得的执行计划信息丰富。

其实如果不考虑包的使用，单独将查看 SQL 语句的执行计划使用 SET AUTOTRACE 实现似乎更方便，如实例 17-20 所示，通过 SET AUTOTRACE 获得一个 SQL 语句的执行计划。

实例 17-20　通过 SET AUTOTRACE 获得 SQL 语句的执行计划。

```
SQL> set autotrace on explain
SQL> SELECT  e1.first_name, e1.last_name, j.job_id, sum(e2.salary) total_sal
  FROM employees e1, employees e2, job_history j
  WHERE e1.employee_id = e2.manager_id
    AND e1.employee_id = j.employee_id
    AND e1.hire_date = j.start_date
  GROUP BY e1.first_name, e1.last_name, j.job_id
  ORDER BY total_sal;  2    3    4    5    6    7

FIRST_NAME           LAST_NAME                 JOB_ID      TOTAL_SAL
-------------------- ------------------------- ---------- ----------
Michael              Hartstein                 MK_REP           6000
Lex                  De Haan                   IT_PROG          9000


Execution Plan
----------------------------------------------------------
Plan hash value: 4189704726

---------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                          |   106 |  5936 |    10  (30)| 00:00:01 |
|   1 |  SORT ORDER BY                |                          |   106 |  5936 |    10  (30)| 00:00:01 |
|   2 |   HASH GROUP BY               |                          |   106 |  5936 |    10  (30)| 00:00:01 |
|   3 |    NESTED LOOPS               |                          |       |       |            |          |
|   4 |     NESTED LOOPS              |                          |   106 |  5936 |     8  (13)| 00:00:01 |
|*  5 |      HASH JOIN                |                          |   106 |  3710 |     7  (15)| 00:00:01 |
|   6 |       TABLE ACCESS FULL       | EMPLOYEES                |   107 |  2889 |     3   (0)| 00:00:01 |
|   7 |       TABLE ACCESS FULL       | EMPLOYEES                |   107 |   856 |     3   (0)| 00:00:01 |
|*  8 |      INDEX UNIQUE SCAN        | JHIST_EMP_ID_ST_DATE_PK  |     1 |       |     0   (0)| 00:00:01 |
|   9 |     TABLE ACCESS BY INDEX ROWID| JOB_HISTORY             |     1 |    21 |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("E1"."EMPLOYEE_ID"="E2"."MANAGER_ID")
   8 - access("E1"."EMPLOYEE_ID"="J"."EMPLOYEE_ID" AND "E1"."HIRE_DATE"="J"."START_DATE")
```

17.1.4　DBMS_HPROF 包

该包是 Oracle 默认安装的包，只要 DBA 具有相应的执行权限即可，在服务器上提供一个目录用于写入信息，为了使用 DBMS_HPROF，最好执行如下操作，这样任何用户都可以使用该包来完成 PL/SQL 程序的配置。

该包的结构如实例 17-21 所示。

实例 17-21　包 DBMS_HPROF 的结构。

```
SQL> connect sys/oracle as sysdba
Connected.
```

```
SQL> desc dbms_hprof
FUNCTION ANALYZE RETURNS NUMBER
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 LOCATION                       VARCHAR2                IN
 FILENAME                       VARCHAR2                IN
 SUMMARY_MODE                   BOOLEAN                 IN     DEFAULT
 TRACE                          VARCHAR2                IN     DEFAULT
 SKIP                           BINARY_INTEGER          IN     DEFAULT
 COLLECT                        BINARY_INTEGER          IN     DEFAULT
 RUN_COMMENT                    VARCHAR2                IN     DEFAULT
 PROFILE_UGA                    BOOLEAN                 IN     DEFAULT
 PROFILE_PGA                    BOOLEAN                 IN     DEFAULT
PROCEDURE START_PROFILING
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 LOCATION                       VARCHAR2                IN     DEFAULT
 FILENAME                       VARCHAR2                IN     DEFAULT
 MAX_DEPTH                      BINARY_INTEGER          IN     DEFAULT
 PROFILE_UGA                    BOOLEAN                 IN     DEFAULT
 PROFILE_PGA                    BOOLEAN                 IN     DEFAULT
PROCEDURE STOP_PROFILING
```

显然该包包含一个函数 ANALYZE 和一个过程 START_PROFILING。为了演示如何使用它们，我们先进行如下工作，使得所有用户可以使用该包，如实例 17-22 所示。

实例 17-22　创建目录对象并授权。

```
SQL> connect sys/oracle as sysdba
Connected.
SQL> grant execute on dbms_hprof to public;
Grant succeeded.
SQL> create or replace directory pro_dir as '/u01/app';
Directory created.
SQL> grant read ,write on directory pro_dir to public;
Grant succeeded.
```

为了使用 DBMS_HPROF 来分析执行结果，需要预先安装相关的表，即在 Oracle 的安装目录下运行一个脚本 dbmshptab.sql。该脚本位于目录/rdbms/admin 下，下面开始运行该脚本，如实例 17-23 所示。

实例 17-23　运行脚本 dbmshptab.sql。

```
SQL> @?/rdbms/admin/dbmshptab.sql
drop table dbmshp_runs                     cascade constraints
           *
ERROR at line 1:
ORA-00942: table or view does not exist
drop table dbmshp_function_info            cascade constraints
           *
ERROR at line 1:
ORA-00942: table or view does not exist
drop table dbmshp_parent_child_info        cascade constraints
           *
ERROR at line 1:
ORA-00942: table or view does not exist
drop sequence dbmshp_runnumber
```

```
              *
ERROR at line 1:
ORA-02289: sequence does not exist
Table created.
Comment created.
Table created.
Comment created.
Table created.
Comment created.
Sequence created.
```

在执行完该脚本后会创建表 DBMSHP_FUNCTION_INFO、DBMSHP_PARENT_CHILD_INFO 和 DBMSHP_RUNS，并且创建序列号 DBMSHP_RUNNUMBER。下面查询这些对象的状态，如实例 17-24 所示。

实例 17-24　查询对象的状态。

```
SQL> select object_name ,object_type ,status from dba_objects where object_name
like 'DBMSHP%';
OBJECT_NAME                    OBJECT_TYPE         STATUS
------------- --------------------------------- ---------------------
DBMSHP_FUNCTION_INFO            TABLE               VALID
DBMSHP_PARENT_CHILD_INFO        TABLE               VALID
DBMSHP_RUNNUMBER                SEQUENCE            VALID
DBMSHP_RUNS                     TABLE               VALID
```

下面创建两个过程：count_employees 和 count_dept，如实例 17-25 所示。

实例 17-25　创建过程 count_employees。

```
SQL>create or replace procedure count_employees
  2   (mgr_id in number)
  3  as
  4   var_count   number;
  5  begin
  6    select count(*)
  7    into var_count from employees
  8     where manager_id=mgr_id;
  9* end;

Procedure created.
```

该过程的输入参数为 NUMBER 类型的数值，用于计算当前 MANAGER_ID 为过程输入数值的员工数量。创建过程 count_dept，如实例 17-26 所示。

实例 17-26　创建过程 count_dept。

```
SQL>create or replace procedure count_dept
  2  (dept_id in number)
  3  as
  4  var_count number;
  5  begin
  6   select count(*) into var_count from employees
  7  where department_id=dept_id;
  8* end;

Procedure created.
```

上述过程用于计算表 EMPLOYEES 中满足特定部门号的员工数量，过程的输入参数为员工的部门号。下面我们继续创建过程，其作用是调用上面创建的两个过程：count_employees 和 count_dept，如实例 17-27 所示。

实例 17-27　创建过程 call_dept_employees。

```
SQL> create or replace procedure call_dept_employees
  2  as
  3  begin
  4    for i in (select distinct manager_id from employees)
  5      loop
  6        count_employees(i.manager_id);
  7      end loop;
  8   for i in (select distinct department_id from employees)
  9      loop
 10        count_dept(i.department_id);
 11      end loop;
 12* end;

Procedure created.
```

以上部分通过三个实例创建了三个过程：count_employees、count_dept 和 call_dept_employees，其中最后一个过程 call_dept_employees 调用了前两个过程。下面我们使用 START_PROFILING 函数启动分层配置器，如实例 17-28 所示。

实例 17-28　使用 START_PROFILING 函数启动分层配置器。

```
SQL>begin
  2  dbms_hprof.start_profiling(
  3    location => 'PRO_DIR',
  4    filename =>'pro.txt');
  5  call_dept_employees;
  6  dbms_hprof.stop_profiling;
  7* end;

PL/SQL procedure successfully completed.
```

这是一个匿名过程，它的作用是完成对过程 call_dept_employees 的配置，然后运行函数 ANALYZE，从而分析原始数据并存储在执行脚本 dbmshptab.sql 创建的表中。下面运行函数 ANALYZE，注意必须使用具有访问脚本 dbmshptab.sql 创建的表权限的用户名登录，否则会报错，如实例 17-29 所示。

实例 17-29　运行函数 ANALYZE（非权限用户）。

```
SQL> set serveroutput on;
declare
 var_runid  number;
 begin
  var_runid := dbms_hprof.analyze(
         location => 'PRO_DIR',
         filename =>'pro.txt',
         run_comment =>' on testing!');
  dbms_output.put_line('var_runid is : '||var_runid);
end;
/SQL>  2    3    4    5    6    7    8    9   10
```

```
declare
*
ERROR at line 1:
ORA-00942: table or view does not exist
ORA-06512: at "SYS.DBMS_HPROF", line 317
ORA-06512: at line 4
```

上述过程在 HR 模式下运行，因为无法访问用户 SYS 调用脚本 dbmshptab.sql 创建的表，所以报错，下面使用 SYS 用户登录，重新执行上面的过程，如实例 17-30 所示。

实例 17-30　运行函数 ANALYZE（权限用户）。

```
SQL> connect sys/oracle as sysdba
Connected.
SQL> declare
  2   var_runid  number;
  3   begin
  4    var_runid := dbms_hprof.analyze(
  5            location => 'PRO_DIR',
  6            filename =>'pro.txt',
  7            run_comment =>' on testing!');
  8    dbms_output.put_line('var_runid is : '||var_runid);
  9* end;
SQL> /
var_runid is : 2

PL/SQL procedure successfully completed.
```

通过上面的分析，可获得过程的运行编号 RUNID，使用该编号可以查询表，获得调用层次信息，也可以通过如下方式获得 RUNID，如实例 17-31 所示。

实例 17-31　通过 DBMSHP_RUNS 获得 RUNID。

```
SQL> col run_comment for a20
SQL> col run_timestamp for a30
SQL>select runid,run_timestamp,total_elapsed_time,run_comment
  2* from dbmshp_runs

 RUNID  RUN_TIMESTAMP          TOTAL_ELAPSED_TIME      RUN_COMMENT
---------- ------------- ------------------ --------------------
         1 29-SEP-12 11.33.43.051172 AM    19021                 on testing!
         2 29-SEP-12 11.33.47.649948 AM    19021                 on testing!
```

通过 RUNID 继续查询表 DBMSHP_FUNCTION_INFO，目的是获得过程 call_dept_ employees 的 SYMBOLID，通过这个值继续查询表 DBMSHP_FUNCTION_INFO 和表 DBMSHP_PARENT_CHILD_INFO，以获得查询的层次性信息。

下面查询表 DBMSHP_FUNCTION_INFO，以获得 SYMBOLID，如实例 17-32 所示。

实例 17-32　查询表 DBMSHP_FUNCTION_INFO 获得 SYMBOLID。

```
SQL> l
  1  select symbolid,owner,module,type,function
  2* from dbmshp_function_info where runid=2 order by 1
SQL> /

 SYMBOLID OWNER      MODULE      TYPE            FUNCTION
```

```
----- ---------- - --------------- ------------------------------
1  HR        CALL_DEPT_EMPLOYEES  PROCEDURE     CALL_DEPT_EMPLOYEES
2  HR        COUNT_DEPT            PROCEDURE      COUNT_DEPT
3  HR        COUNT_EMPLOYEES      PROCEDURE     COUNT_EMPLOYEES
4  SYS       DBMS_HPROF           PACKAGE BODY    STOP_PROFILING
5  HR        CALL_DEPT_EMPLOYEES  PROCEDURE     __sql_fetch_line4
6  HR        CALL_DEPT_EMPLOYEES  PROCEDURE     __sql_fetch_line8
7  HR        COUNT_DEPT            PROCEDURE       __static_sql_exec_line6
8  HR        COUNT_EMPLOYEES      PROCEDURE      __static_sql_exec_line6

8 rows selected.
```

从输出可以知道，第一行说明 CALL_DEPT_EMPLOYEES 的 SYMBOLID 为 1，结合 RUNID 为 2，执行如下复合查询，以获得过程的内部调用层次关系，如实例 17-33 所示。

实例 17-33　查询过程的内部调用层次关系。

```
SQL> select rpad(' ',level*2,' ') || f.owner ||'.'||f.module as name,
  2  f.function,
  3  pi.subtree_elapsed_time,
  4  pi.function_elapsed_time,
  5  pi.calls
  6  from dbmshp_parent_child_info pi
  7   join dbmshp_function_info f on pi.runid = f.runid and
  8  pi.childsymid = f.symbolid
  9  where pci.runid=2
 10  connect by prior childsymid = parentsymid
 11* start with pi.parentsymid = 1
SQL> /

NAME   FUNCTION  SUBTREE_ELAPSED_TIME  FUNCTION_ELAPSED_TIME CALLS
- -------------------- -------------------- --------------------- -----
HR.COUNT_DEPT     __static_sql_exec_li ne6         2851        2851     12
HR.COUNT_DEPT       COUNT_DEPT             3128        277       12
HR.COUNT_DEPT     __static_sql_exec_li ne6         2851        2851     12
HR.COUNT_EMPLOYE ES __static_sql_exec_li ne6      3823         3823     19
HR.COUNT_EMPLOYEES COUNT_EMPLOYEES         4526        703       19
HR.COUNT_EMPLOYEES __static_sql_exec_li ne6      3823        3823     19
HR.CALL_DEPT_EMPLO YEES __sql_fetch_line4       7859        7859      1
HR.CALL_DEPT_EMPLO YEES __sql_fetch_line8       2847        2847      1

8 rows selected.
```

表记录了 PL/SQL 过程的执行和调试信息，通过分析可得出 PL/SQL 函数的内部调用关系。

17.2　警告日志文件包

警告日志文件是 DBA 每天都要查看的文件，其中记录了重要的需要 DBA 分析的错误消息，如数据文件离线无法读取、重做日志文件丢失等。当然 DBA 可以逐行分析警告文件以获得必要的 ALERT 信息，但是这种方式显然需要耗费更多的时间与精力，如果可以自动化监控警告日志的内容，将每天的警告日志发送到 DBA 的信箱，这样 DBA 就可以集中精力处理每天的警告记录，而没必要从警告日志文件中搜索后再分析特定的错误提示。当然对于警告文件的获取有多种方式，例

如使用外部表就是其中一种。

17.2.1 设计外部表

可以创建一个外部表来管理警告日志文件，这样就可以通过外部表来存储警告文件信息，DBA可以通过读表的方式来分析警告文件，如实例 17-34 所示。

实例 17-34 创建目录对象 ALERT_DIR。

```
SQL> create or replace directory alert_dir
  2  as '/u01/app/oracle/diag/rdbms/orcl1/orcl1/trace';

Directory created.
```

下面可以通过 DBMS_METADATA 包的 GET_DDL 过程获得目录对象的定义，使用该过程可以获得数据字典中记录的所有数据库对象的定义，目录对象的定义如实例 17-35 所示。

实例 17-35 获得目录对象定义。

```
SQL>  select dbms_metadata.get_ddl('DIRECTORY','ALERT_DIR') FROM DUAL;

DBMS_METADATA.GET_DDL('DIRECTORY','ALERT_DIR')
----------------------------------------------------------------
   CREATE OR REPLACE DIRECTORY "ALERT_DIR" AS '/u01/app/oracle/diag/rdbms/orcl1
```

在创建了目录对象后，因为我们已经知道告警日志文件的目录以及文件名称，所以开始创建一个外部表（具体语法请读者参考相应数据库版本的 SQL Reference），如实例 17-36 所示。

实例 17-36 创建一个外部表 ALERT_DIAG。

```
SQL>create table alert_diag
  2  (
  3   alert_msg  varchar2(1000)
  4  )
  5  organization external
  6   (type oracle_loader
  7    default directory "ALERT_DIR"
  8    access parameters
  9    (records delimited by newline
 10    )
 11   location
 12    ('alert_orcl1.log')
 13   )
 14  reject limit unlimited
 15* /

Table created.
```

该外部表通过 SQL-LOADER 加载文件中的数据，当加载数据时每读一行数据就存入外部表。在外部表创建成功后，下面验证该表的结构，如实例 17-37 所示。

实例 17-37 验证外部表结构。

```
SQL> desc alert_diag;
 Name                                       Null?    Type
```

```
 ---------------------- --------- ----------------------------
 ALERT_MSG                                   VARCHAR2(1000)
```

外部表创建成功后，必须使用 SELECT 语句查询，否则不能判定外部表是否加载数据成功，即表结构创建了，但是有可能由于访问参数的错误造成无法读取文件中的数据，如实例 17-38 所示，虽然表创建成功，但是仍存在错误。

实例 17-38　重新创建外部表（有错误）。

```
SQL> l
  1  create table alert_diag
  2  (
  3   alert_msg  varchar2(1000)
  4  )
  5  organization external
  6   (type oracle_loader
  7    default directory "ALERT_DIR"
  8    access parameters
  9    (records delimited by newline
 10     reject rows with all fields (alert_msg (1:1000) char(1000))
 11    )
 12   location
 13    ('alert_orcl1.log')
 14   )
 15  reject limit unlimited
 16* /

Table created.
```

此时表创建成功，下面查询其结构。

```
SQL> desc alert_diag;
 Name                                    Null?    Type
 ---------------------- --------- ----------------------------
 ALERT_MSG                                   VARCHAR2(1000)
```

当查询该表中是否存在数据时，由于无法读取文件而报错，所以必须注意虽然表创建成功，但未必意味着查询一定有结果，因为外部表最终的数据是放在文件中，外部表仅仅是便于用户使用 SQL 语句操纵文件数据的一种方式。下面是查询的错误提示，如实例 17-39 所示。

实例 17-39　读取外部表的错误提示。

```
SQL> select * from alert_diag;
select * from alert_diag
*
ERROR at line 1:
ORA-29913: error in executing ODCIEXTTABLEOPEN callout
ORA-29400: data cartridge error
KUP-00554: error encountered while parsing access parameters
KUP-01005: syntax error: found "reject": expecting one of: "badfile,
byteordermark, characterset, column, data, delimited, discardfile,
disable_directory_link_check, fields, fixed, load, logfile, language,
nodiscardfile, nobadfile, nologfile, date_cache, preprocessor, readsize,
string, skip, territory, variable"
KUP-01007: at line 2 column 4
```

之后就可以像查询普通表一样，通过外部表来查询警告文件中的信息了，代码如下所示。

```
SQL>  select * from alert_diag  where alert_msg like 'ORA-%' and rownum<6;

ALERT_MSG
-------------------------------------------------------------
ORA-07445: exception encountered: core dump [nttaddr2bnd()+2284] [SIGSEGV]
[ADDR
:0x0] [PC:0xA75CE80] [Address not mapped to object] []

ORA-1109 signalled during: ALTER DATABASE CLOSE NORMAL...
ORA-07445: exception encountered: core dump [nttaddr2bnd()+2284] [SIGSEGV]
[ADDR
:0x0] [PC:0xA75CE80] [Address not mapped to object] []

ORA-00313: open failed for members of log group 1 of thread 1
ORA-00317: online log 1 thread 1: '/u01/app/oracle/oradata/orcl1/redo01.log'
```

17.2.2 设计警告文件监控包

在上例中，我们创建了外部表 alert_diag，从而可以通过进一步创建监控包来分析监控警告文件，具体来说我们可以通过创建一个 PROGRAM 来调用这个包，并分析警告文件内容，然后创建一个 JOB，将该 JOB 与具体执行窗口关联，这样我们就创建了一个自动的警告文件监控任务。根据实际的监控需要，就可以改写 PROGRAM，从而完成需要的逻辑分析。下面创建该包的一个过程，然后通过该过程分析结果，如果发现 ORA 错误，就将错误消息保存下来，并提醒用户。

下面先创建一个表，用于存储调用过程时存入数据的表。将告警日志中的所有 ORA 错误消息都存入一个表。定期运行这个过程，就可以记录错误消息，如实例 17-40 所示。

实例 17-40 创建表 alert_t。

```
SQL> create table alert_t (mydate date,mymsg varchar2(1000));

Table created.
```

下面是警告文件包的创建过程 monit_alert，如实例 17-41 所示。

实例 17-41 创建警告文件过程 monit_alert。

```
SQL>create or replace procedure  monit_alert
  2  as
  3   cursor c1 is
  4     select alert_msg
  5       from alert_diag where alert_msg like 'ORA-%';
  6
  7   l_msg_text     varchar2(32767);
  8
  9  begin
 10    open c1;
 11
 12  loop
 13    fetch c1 into l_msg_text;
 14      insert into alert_t values(sysdate,l_msg_text);
 15  exit when c1%notfound;
 16  end loop;
 17
 18   close c1;
```

```
 19* end;
SQL> /

Procedure created.
```

该过程的逻辑很简单，就是通过游标将所有错误消息 ORA 都提取出来，然后存入一个表中。这样用户就可以根据表查找所需要的错误提示，当然可以查询用户关心的特定错误类型，或者一旦查询到特定的错误类型就通过邮件通知用户等，这些就不一一介绍了。执行过程后开始查询表 alert_t 的结果，如实例 17-42 所示。

实例 17-42　查询表 alert_t。

```
SQL> select * from alert_t where rownum<4;

MYDATE    MYMSG
--------- ---------------------------------------------------------
08-OCT-12 ORA-07445: exception encountered: core dump [nttaddr2bnd()+2284]
[SIGSEGV] [ADDR:0x0] [PC:0xA75CE80] [Address not mapped to object] []
08-OCT-12 ORA-1109 signalled during: ALTER DATABASE CLOSE NORMAL...
08-OCT-12 ORA-07445: exception encountered: core dump [nttaddr2bnd()+2284]
[SIGSEGV] [ADDR:0x0] [PC:0xA75CE80] [Address not mapped to object] []
```

这个查询结果显示了某一天所有的错误类型，这里我们没有过滤相同类型的错误消息，有兴趣的读者可以修改过程 monit_alert 以实现该功能。

17.3　数据库维护包

17.3.1　备份监控包

对于数据库的备份任务，DBA 都是通过配置自动执行的脚本来完成的，例如在 Windows 下使用计划任务等，对于备份过程可使用 LOG 来记录备份过程的信息，如果发生错误，可以查看日志。但是如果由于其他原因造成备份没有被执行，此时就无法通过日志发现问题，因为备份任务根本没有执行，所以就需要采取其他方式解决备份没有执行的问题。

在 Oracle 中当使用热备份或者使用 RMAN 工具备份时，这些备份信息会记录在动态性能视图中，使用视图可以清楚地看到每一条备份的详细记录，例如视图 v$backup_set，该视图记录了完整的备份记录。下面是该视图的列属性。

- BACKUP_TYPE：备份中的文件类型，如果包含归档日志文件，则其值为 L；如果是数据文件全备份，则其值为 D；如果是增量备份，则其值为 I。
- COMPLETION_TIME：备份任务完成的时间。

下面执行一个全库备份操作：

```
RMAN> backup database;
```

然后，执行如下的全库压缩备份，并且包含了归档日志，如实例 17-43 所示。

实例 17-43　执行全库压缩备份，含归档日志。

```
RMAN> backup as compressed backupset database plus archivelog delete all input;


Starting backup at 07-OCT-12
current log archived
using channel ORA_DISK_1
channel ORA_DISK_1: starting compressed archived log backup set
channel ORA_DISK_1: specifying archived log(s) in backup set
input archived log thread=1 sequence=63 RECID=36 STAMP=796087894
channel ORA_DISK_1: starting piece 1 at 07-OCT-12
```

此时，在动态性能视图 v$backup_set 中记录了上面执行的备份，查询如实例 17-44 所示。

实例 17-44　查询备份记录信息。

```
SQL> select recid,backup_type,to_char(completion_time,'yyyy-mm-dd hh24:mi:
ss') completion_time,incremental_level from v$backup_set;

     RECID B COMPLETION_TIME     INCREMENTAL_LEVEL
     ------------ ---------------- --------------------------------
        1 D 2012-10-07 22:35:29
        2 D 2012-10-07 22:35:44
        3 L 2012-10-07 23:31:36
        4 D 2012-10-07 23:34:40
        5 D 2012-10-07 23:34:48
        6 L 2012-10-07 23:34:52

6 rows selected.
```

显然，输出显示了备份类型有 D 和 L，D 代表最开始的全库备份，而 L 代表全库包含归档日志的压缩备份。

下面主要依据备份完成时间 COMPLETION_TIME 监控备份是否按期完成，但是这个方法又需要考虑我们的备份周期。假定每天晚上 12 点执行一次全库备份，则离当前最近的一次备份不会超过一天，如果超过一天则有理由认为上次的备份没有成功，此时就需要 DBA 去分析这个原因了。下面是依据这个思路编写的一个监控过程，用于监控第一个备份是否顺利完成，如实例 17-45 所示。

实例 17-45　创建备份监控过程 monit_backup。

```
procedure monit_backup
is
   message varchar(60);
   monit_date date;
begin

  select sysdate||extract(day from (systimestamp-completion_time)||'days'||
                extract(hour from (systimestamp-completion_time)||'hours'||
                extract(hour from (systimestamp-completion_time)||'munites
                ago',sysdate
      into message,monit_date
  from v$backup_set b
  where completion_time=
     (select max(completion_time) from v$backup_set
      where incremental_level=b.incremental_level and
```

```
        backup_type=b.backup_type) and

    backup_type='D' and incremental_level is null and
   sysdate- completion_time >1;

  insert into backup_t values(message,monit_date);
  end;
  /
```

在上例中，查询了距离现在最近的一次备份时间，如果该时间超过一天，则记录这个事件，并且我们查看的是数据文件全库备份的备份类型。一旦发生这样的事情，则将记录写入一个表，当然这里也可以启动一个 EMAIL 功能，将这些信息发送到 DBA 邮箱中。

17.3.2　表空间监控包

表空间是存储数据的逻辑对象，如果表空间的空间不够，而数据文件又不能自动扩展，此时数据库就会报错，事务无法进行下去。如果使用的数据文件能够自动扩展，那么我们就不必担心这个问题，但是使用表空间监控包的功能依然可以提供给我们一些有价值的信息，那么如何获得表空间的使用呢？首先，使用数据字典 dba_free_space 获得查询对应表空间的空闲空间，然后通过数据字典 dba_data_files 获得表空间已经分配的空间和潜在的扩展空间。通过如下两个数据字典，就可以获得计算表空间的使用率。

- select tablespace_name,sum(bytes) free from dba_free_space;
- select tablespace_name,sun(bytes) allocated,sum(maxbytes) potential from dba_data_files;

下面查询一下表空间的使用率，如实例 17-46 所示。

实例 17-46　查询表空间的使用率。

```
  SQL>select to_char(sysdate,'yyyy-mm-dd hh24:mi:ss') mydate,a.tablespace_
    name, to_char(round(1-(free+potential)/(allocated+potential),4)*100) rate
    2     from
    3     (select tablespace_name,sum(bytes) free from dba_free_space group by
           tablespace_name) f,
    4    (select tablespace_name,sum(bytes) allocated,sum(maxbytes) potential
         from dba_data_files group by
    5  tablespace_name) a
    6*  where a.tablespace_name=f.tablespace_name
  SQL> /

  MYDATE              TABLESPACE_NAME            RATE
  ------------------------------ ----------------------------------------
  2012-10-04 22:13:55 SYSAUX                     1.32
  2012-10-04 22:13:55 UNDOTBS1                   .03
  2012-10-04 22:13:55 USERS                       .01
  2012-10-04 22:13:55 SYSTEM                     2.02
  2012-10-04 22:13:55 EXAMPLE                    .24
```

从上述查询可以知道表空间 SYSTEM 的使用率最高，但是由于是新库，只能获得所有空间的 2.02%。

依据上面的分析和查询结果设计一个监控表空间的包，该包包含一个过程，其名称为

monit_tbs，如实例 17-47 所示。

实例 17-47　创建监控表空间的存储过程。

```
SQL>create or replace procedure monit_tbs
is
 var_date varchar2(20);
 var_tbs  varchar2(20);
 var_rate varchar2(20);
 cursor cur_tbs
 is
    select to_char(sysdate,'yyyy-mm-dd hh24:mi:ss') mydate,a.tablespace_name,
    to_char(round(1-(free+potential)/(allocated+potential),4)*100) rate
    from
    (select tablespace_name,sum(bytes) free from dba_free_space group by
     tablespace_name) f,
  (select tablespace_name,sum(bytes) allocated,sum(maxbytes) potential from
  dba_data_files group by
tablespace_name) a
 where a.tablespace_name=f.tablespace_name;
begin
 open cur_tbs;
loop
  fetch cur_tbs into var_date,var_tbs,var_rate;
   if var_rate>80
  insert into storage_t values (var_tbs||' at '||var_date,rate);
exit when cur_tbs%notfound;
end loop;
close cur_tbs;
end;
/
```

创建一个游标，将查询的结果存入表 storage_t，该表含有两列 message 和 value。该游标在向表插入数据时需要判断 value 的值是否大于一个门限值，如果大于，则存入下面的数据：“message= var_tbs||' at '||var_date '”，而 value= to_char(round(1-(free+potential)/(allocated+potential),4)*100)。

如果发生表空间使用空间超过了设置的门限值，则可以从表中发现如实例 17-48 所示的消息。

实例 17-48　查询表 storage_t，以获得告警信息。

```
SQL>select event,value from storage_t;
message                                     value
-------------------------------------------------- ---------------
USERS at 2012-10-04 22:13:55    81
```

该表中记录的值 81 说明表空间的值超过了额定值，需要注意表空间的大小了。如果当前的表空间没有上限，而是数据文件自动扩展，此时的告警记录同样会提示我们数据的增长很快，需要引起注意了。

17.3.3　归档目录监控包

当数据库处于归档模式下时，Oracle 会自动重做日志的归档，这样就可以实现对数据库的完全恢复，但是如果归档目录的空间不足，此时重做日志的重做记录就无法实现归档保存，Oracle 会挂起数据库。其实这个原理也很简单，既然 Oracle 将数据库处于归档模式，那么自然是保证在

发生介质故障时，可以完全恢复数据库。但是此时由于归档目录已满，没有空间可以存储归档日志，所以 Oracle 将停止对客户的服务，以防止产生更多新的重做日志，而这些重做日志又无法归档。从数据库结构的角度可以这样理解，重做日志组是循环使用的，既然当前的重做日志组无法归档，那么 Oracle 也不允许切换重做日志，此时数据库只能“停”在那里，即挂起数据库。

首先设置数据的归档目录，并且设置该归档目录的最大使用空间限额，如实例 17-49 所示。

实例 17-49　设置归档目录的最大空间限额。

```
SQL> alter system set log_archive_dest_1='location=/u01/backup quota_size=
200m' scope=spfile;

System altered.
```

接着通过 v$archive_dest 视图获取归档目录的空间使用情况，该视图会动态记录归档目录的空间情况，如实例 17-50 所示。

实例 17-50　查询归档目录的空间情况。

```
SQL> col destination for a40
SQL> select destination,quota_size,quota_used from v$archive_dest
  2  where destination is not null;

DESTINATION                          QUOTA_SIZE   QUOTA_USED
---------------------------- --------------------- --------------------------
/u01/backup                                209715200      0
```

在上例中可以看出已设置了归档目录，修改了参数 log_archive_dest_1，并设置归档目录的上限为 200M。下面讨论一下归档目录的设置要求。

若没有指定 log_archive_dest 参数，则默认位于 db_recovery_file_Dest 目录下。如果只需要设置本地存储，则需要设置 log_archive_Dest 和 log_archive_dest_duplex。如果需要设置远程归档和本地归档，则使用 log_archive_dest_n 参数设置。

在设置了归档目录后，可以通过视图查询归档空间的使用情况，这也是创建归档目录监控过程的核心视图，如实例 17-51 所示。

实例 17-51　通过 v$archive_dest 查询归档空间的使用情况。

```
SQL> select dest_name,destination,quota_size,quota_used
  2  from v$archive_dest
  3  where quota_size>0;

DEST_NAME            DESTINATION       QUOTA_SIZE QUOTA_USED
------------------------ ----------- ---------- ----------
LOG_ARCHIVE_DEST_1    /u01/backup         209715200     174
```

通过上面的查询可以知道归档目录/u01/backup 的分配空间为 209715200bytes，已经使用了 174bytes，所以通过组合计算可以获得归档空间的使用情况，如实例 17-52 所示。

实例 17-52　计算可归档空间的使用率。

```
SQL>select to_char(sysdate,'yyyy-mm-dd hh24:mi:ss'),dest_name||':'||destination,
  2  round(1-(quota_size-quota_used)/(quota_size),8)*100 rate from v$archive_ dest
  3* where quota_size>0
```

```
TO_CHAR(SYSDATE,'YY   DEST_NAME||':'||DESTINATION          RATE
---------------- ------------------------------------- ----------
2012-10-06 11:17:08       LOG_ARCHIVE_DEST_1:/u01/backup     .001558
```

从上面的查询可以知道此时归档目录的空间使用率，这个值目前虽然很低，但是随着事务数量的增长，归档数据会越来越多，所以需要监控该归档目录的空间使用情况。下面是要创建的过程，如实例 17-53 所示。

实例 17-53　创建过程 monit_arch。

```
SQL> create or replace procedure monit_arch
  2  is
  3    message varchar2(50);
  4    value   varchar2(20);
  5    begin
  6       select dest_name||':'||destination||' at '||to_char(sysdate,'yyyy-
  7       mm-dd hh24:mi:ss') mesg,round(1-(quota_size- quota_used)/
  8       (quota_size),8)*100 rate into message,value
  9           from v$archive_dest  where quota_size>0;
 10      if value>85 then
 11        insert into arch_t (message,value);
 12      end if;
 13* end;
```

monit_arch 作为数据库维护包的一个过程，该过程的逻辑很简单，使用数据字典视图 v$archive_dest 获得归档空间的使用情况，然后计算一个阀值，如果空间使用率超过这个阀值就写入表 arch_t 中一条告警信息，查询的结果如实例 17-54 所示。

实例 17-54　查询表 arch_t 中的告警消息。

```
SQL>Select message,value
    from arch_t;

MESSAGE                                                 VALUE
------------------------------------------------- --------------
LOG_ARCHIVE_DEST_1:/u01/backup at 2012-10-06 11:32:22       87
```

上例查询的结果说明，已经发生一次告警，且当前的归档目录空间已经使用了 87%，超过了 85%的阀值，需要 DBA 为其增加空间了，或者将归档备份出来以释放空间。

17.4　历史数据包

历史数据是 Oracle 数据库中需要监控的历史分析的数据，这些数据包括数据库中全部数据的大小（近似大小），例如当天某个时刻连接到数据库的会话数量，显然，过多的会话会耗费数据库过多的内存以及 CPU 资源，需要我们监控会话数量值，以提示是否超过我们允许的最大会话数，当然这个数值还受到购买的服务限制。下面先看一下如何监控数据库大小，然后创建历史数据包，这个包有两个过程：一个负责监控数据库的大小；另一个负责监控会话数的多少。

17.4.1　监控数据库的大小

数据库中最重要的逻辑存储单位是段 SEGMENTS，它介于表空间和区 EXTENT 之间，这些段包括表段、索引段等。通过对段数据量的查询，基本可以确定数据库中数据的增长情况，从而确定整个数据库中数据的大小和发展趋势。下面我们查询当前数据库的所有段中数据的大小，如实例 17-55 所示。

实例 17-55　查询当前数据库的所有段中的数据。

```
SQL> select sysdate,to_char(round(sum(bytes)/(1024*1024),2)) from dba_segments;

SYSDATE        TO_CHAR(ROUND(SUM(BYTES)/(1024*1024),2))
-------------- ----------------------------------------
03-10月-12      817.06
```

上面的查询结果是当前库的所有段的大小，其实在所有段中应该是表段的数据量最大，否则就是有问题的，如实例 17-56 所示，查询数据字典 DBA_SEGMENTS，从中按照类型来查询不同段类型的段数据量大小。

实例 17-56　查询当前库中不同段类型的数据量。

```
SQL> select segment_type,sum(bytes) from dba_segments group by segment_type;

SEGMENT_TYPE        SUM(BYTES)
------------------ ----------
LOBINDEX             36175872
INDEX PARTITION        22413312
TABLE PARTITION        24641536
NESTED TABLE             720896
ROLLBACK                393217
LOB PARTITION            65536
LOBSEGMENT            98697217
INDEX                215154688
TABLE                400424960
CLUSTER               30408704
TYPE2 UNDO             27656192

已选择 11 行。
```

从查询可以知道，还是 TABLE 类型的段数据量最大。下面创建一个过程，用于监控数据库的大小，每天执行一次，记录当前库的大小，并存入表 dbsize_t 中，该表存储了过程执行时间以及当前库中段的总数据量，如实例 17-57 所示。

实例 17-57　创建过程 monit_dbsize。

```
SQL> create or replace procedure monit_dbsize
  2  is
  3    message varchar2(50);
  4    value   varchar2(20);
  5    begin
  6        select to_char(sysdate,'yyyy-mm-dd hh24:mi:ss') moni_date,
  7          to_char(round(sum(bytes)/(1024*1024),2)) size
  8          into message,value
  9          from dba_segments;
```

```
  10      insert into dbsize_t (message,value);
  11* end;
```

以上代码创建了监控数据库大小的过程，虽然这个计算结果不会十分准确，但是还是十分有价值的。例如每隔一周就可以查询该表，以获得每天的数据库中的数据量，同时一个很重要的作用是通过这些值可以获得每天的数据增长量，这样就可以有效评估整个数据库在一段时间内的数据量会增加大致多少，从而有效维护数据库，如合理设置表空间大小等。

17.4.2 监控会话数

通过 SESSION 可以知道会话占用的系统资源，因为 Oracle 服务器首先会分配一个服务器进程并且会分配特定的内存区域，所以如果存在大量无用的连接，将白白占用这些资源，可以使用 v$session 动态性能视图查看当前实例的会话信息，如实例 17-58 所示。

实例 17-58 查看当前实例的会话信息。

```
SQL> select sid,serial#,username,status,process,program,terminal from v$session
  2  where program not like 'ORACLE%';
SID  SERIAL# USERNAME   STATUS   PROCESS     PROGRAM   TERMINAL
--- ---------- ---------- -------- ------------ --- ----------------- --------
 158  54      SYS         ACTIVE   3856:3976    sqlplus.exe     BJ-BDC-JKZ-DCN
```

上例用于查找非 Oracle 自身的会话记录，其中 SID 和 SERIAL#可以确定一个会话，如果该会话占用过多资源而不释放，必要的时候可以关闭该会话。

```
SQL>alter system kill session '158,54';
```

这里对于会话的监控主要是对会话数量的监控，如果数据库性能出现问题，而且推断问题可能出现在过多的连接会话上，此时就可以查看当前的连接数量（不是很准确，但具有指导意义），如果通过经验判断认为超过某个会话数就会增加数据库性能发生的概率，此时可以设置这个阀值，从而监控数据库实例的连接会话数量，查询获得总的会话数量如实例 17-59 所示。

实例 17-59 查询当前实例的总的会话数。

```
SQL> select to_char(sysdate,'yyyy-mm-dd hh24:mi:ss') ,count(*) from v$session
  2  where program not like 'ORACLE%';

TO_CHAR(SYSDATE,'YY   COUNT(*)
----------------------------------- --------------
2012-10-03 17:50:23         15
```

下面就依据上面的 SQL 语句查询，创建会话数的监控过程代码，如实例 17-60 所示。

实例 17-60 创建会话数的监控过程。

```
SQL>  create or replace procedure monit_sessions
  2    is
  3     var_date varchar2(20);
  4     var_count number;
  5    begin
  6     select to_char(sysdate,'yyyy-mm-dd,hh24:mi:ss') ,count(*)
  7     into var_date,var_count
  8    from v$session
```

```
 9   where program not like 'ORA%';
10   insert into sess_t values(var_date,var_count);
11   commit;
12  end;
13  /

过程已创建。
```

上例创建了过程 monit_sessions，用于监控会话数量，这个过程可以在数据库运行的典型时刻运行，用于记录大负载或者大数据量的情况下的会话数量，也可以每隔 2 个小时记录一次，这样就可以记录每天的平均会话数。通过平均会话数可以大致掌握数据库的业务量。下面执行该过程，并验证过程的执行结果，如实例 17-61 所示。

实例 17-61　验证过程的执行结果。

```
SQL> select * from sess_t;

MYDATE                       MYCOUNT
---------------------------------------- ----------
2012-10-07,07:50:11           7
2012-10-07,08:50:11           8
2012-10-07,09:50:11           9
2012-10-07,10:50:11           12
2012-10-07,11:50:11           7
2012-10-07,12:50:11           20
2012-10-07,13:50:11           9
2012-10-07,17:50:11           17
2012-10-07,15:50:11           12
2012-10-07,17:50:11           17
2012-10-07,17:50:11           8
2012-10-07,18:50:11           11
```

在上例，每隔一小时就调用一次过程 monit_sessions，这样在表 sess_t 中就记录了每个小时的当前数据库实例的会话数量，从而可以依据这个数据来计算每天的平均会话数量，如实例 17-62 所示。

实例 17-62　计算平均会话数量。

```
SQL> select sum(mycount)/count(*) avg from sess_t;

       AVG
------------------------------------------
       11.08
```

17.4.3　资源管理器

Oracle 数据库软件是安装在服务器硬件上的，而一旦有大量用户访问数据库，则数据库资源的管理将成为不可回避的问题，如 CPU、I/O 以及内存资源，所以需要采取某种手段来管理服务器资源，Oracle 使用资源管理器完成数据库服务器资源的管理。

资源管理器只是一个程序包而已，通过这个程序包，DBA 可以为不同的用户指定所能使用的资源的上限。下面介绍几个重要概念。

- 资源用户组：表示具有相同资源需求的用户的集合，在 Oracle 数据库中，只能为资源用户组指定所能使用的资源，不能为单个用户指定。例如一个数据库白天是 OLTP、晚上为批处理业务，因此将白天的进行 OLTP 业务的用户组织起来作为一个资源组（OLTP_GRP），将晚上进行批处理的用户组织起来作为另一个资源组（BATCH_GRP）。
- 资源分配方法：表示某种特定的资源在不同的资源用户组之间如何分配，例如白天 OLTP_GRP 分配的 CPU 资源是 80%，BATCH_GRP 分配的 CPU 资源是 20%，晚上反之。
- 资源计划：资源计划是描述所有的资源如何在所有的用户组之间进行分配的蓝图。Oracle 允许在资源计划里再创建资源子计划。数据库可以有多个资源计划，但是在同一个时刻，只能有一个资源计划是激活的。当用户连接到数据库以后，其对应的 Session 在满足一定条件的基础上，可以在不同的资源组之间进行切换。

Oracle 定义了具体的操作来限制不同的竞争操作如何有效地分配资源。资源管理器允许创建资源计划，这个计划设置了各个消费组可以使用的服务器资源，将需求类似的用户放到一个资源消费组。资源管理器还可以限制会话的最大空闲时间、资源消费组的最大并发会话数，以及限制一个资源消费组的 UNDO 空间。资源管理器可以管理的资源如下所示。

- CPU：指定不同的资源用户组所能使用的 CPU 时间，以能够使用的 CPU 时间占总 CPU 时间的百分比来体现。
- 并行度：指定不同的资源用户组里的用户在执行操作时，同一个资源用户组中的所有的用户所能指定的最大并行度的总额。
- 活动的 Session 个数：限制资源用户组里的用户所能产生的活动 Session 的最大个数总和。活动的 Session 指的是要么消耗 CPU、要么等待 I/O，不活动指的就是既没有消耗 CPU，也没有等待 I/O。
- 产生的 UNDO 数据量：指定资源组里的用户能够产生的 undo 的总量，以 MB 为单位，当达到上限时，资源用户组里的用户就不能再进行 DML 操作。
- 执行某个操作所花的时间：可以配置成当某个操作执行的时间达到上限以后，就将执行该操作的用户切换到其他用户组，从而使得该用户所能使用的资源发生改变。
- 最大空闲时间：表示如果用户空闲超过上限，则切断该用户的连接。

Oracle 也提供了默认的资源设置，它可以满足一般用户的需求，通过动态性能视图 v$resource_limit 查询当前数据库的资源设置情况，如实例 17-63 所示。

实例 17-63　查询当前数据库的资源设置情况。

```
SQL> select resource_name,max_utilization,limit_value from v$resource_limit;

RESOURCE_NAME                 MAX_UTILIZATION LIMIT_VALUE
------------------------------ --------------- --------------------
processes                               24         150
sessions                                29         170
enqueue_locks                            19        2380
enqueue_resources                        36       UNLIMITED
ges_procs                                0          0
ges_ress                                 0       UNLIMITED
```

```
ges_locks                                   0      UNLIMITED
ges_cache_ress                              0      UNLIMITED
ges_reg_msgs                                0      UNLIMITED
ges_big_msgs                                0      UNLIMITED
ges_rsv_msgs                                0         0

RESOURCE_NAME                MAX_UTILIZATION LIMIT_VALUE
---------------------------- --------------- --------------------
gcs_resources                               0         0
gcs_shadows                                 0         0
dml_locks                                  44     UNLIMITED
temporary_table_locks                       0      UNLIMITED
transactions                                4      UNLIMITED
branches                                    0     UNLIMITED
cmtcallbk                                   1     UNLIMITED
sort_segment_locks                          1      UNLIMITED
max_rollback_segments                      11       65535
max_shared_servers                          1      UNLIMITED
parallel_max_servers                        0       3600

已选择 22 行。
```

通过视图查询可以知道数据库包括哪些资源，这些资源值的上限与下限设置，例如 max_shared_servers 表示最多的共享服务器资源设置，它的 MAX_UTILIZATION 为 1，而上限是 UNLIMITED，即没有限制；processes 的 MAX_UTILIZATION 为 24，上限为 150，同时限制了会话数，因为如果是专有连接模式，一个连接就会对应一个数据库服务器进程。

下面通过动态性能视图 v$resource_limit 进行查询。这些资源不包括那些上限为 UNLIMITED 的资源，因为这些资源完全受控于计算机硬件的资源限制，而对于其他上限为非 UNLIMITED 的资源，计算这些资源的使用率就具有指导意义了，例如 processes 的使用率达到了 95%，此时就要考虑是否需要适当增加该参数的值，以允许数据库提供更多的进程服务，当然这也受限于计算机硬件的服务能力。查询代码如实例 17-64 所示。

实例 17-64　查询当前上限为非 UNLIMITED 资源的使用情况。

```
SQL> select resource_name,max_utilization,limit_value from v$resource_limit
where trim(limit_value) !='UNLIMITED';

RESOURCE_NAME                MAX_UTILIZATION LIMIT_VALUE
-------------- --------- -------------------------------
processes                                  24        150
sessions                                   29        170
enqueue_locks                              19        2380
ges_procs                                  0         0
ges_rsv_msgs                               0         0
gcs_resources                              0         0
gcs_shadows                                0         0
max_rollback_segments                     11         65535
parallel_max_servers                       0         3600

已选择 9 行。
```

基于上述的查询基础，再具体查询一下当前一些资源的使用率，这个查询也是我们创建监控过程的一个核心内容，如实例 17-65 所示。

实例 17-65 查询资源的使用率。

```
SQL> select resource_name,(max_utilization/limit_value)*100 rate_used
  2  from v$resource_limit
  3  where trim(limit_value)!='UNLIMITED'
  4  and   trim(limit_value)!='0';

RESOURCE_NAME                  RATE_USED
------------------------------ ----------
processes                           17.5
sessions                            17.4444444
enqueue_locks                        .915254237
max_rollback_segments                .017784924
parallel_max_servers                 .055555556
```

我们将上述的查询 SQL 写入对资源进行监控的过程，以存储那些使用率超过 80% 的资源，从而对那些具有潜在资源受限的资源作出相应的措施，当然也可以设置邮件通知功能，将上述结果发到指定的邮箱。下面是创建的监控资源使用过程，如实例 17-66 所示。

实例 17-66 创建过程 monit_res。

```
SQL> create or replace procedure monit_res
  2  is
  3   var_name varchar2(40);
  4   var_value number(5,3) ;
  5   cursor cur_res
  6   is
  7   select resource_name,(max_utilization/limit_value)*100 rate_used
  8      from v$resource_limit
  9     where trim(limit_value)!='UNLIMITED'
 10      and   trim(limit_value)!='0';
 11
 12  begin
 13   open cur_res;
 14  loop
 15    fetch cur_res into var_name,var_value;
 16    dbms_output.put_line('var_value: '||var_value,3);
 17     if var_value>8  then
 18    insert into res_t values (var_name,var_value);
 19    end if;
 20  exit when cur_res%notfound;
 21
 22  end loop;
 23  close cur_res;
 24  end;
 25  /

过程已创建。
```

在过程 monit_res 中，记录了上限资源不是 UNLIMITED，并且值不是 0 的资源使用率，如果这个使用率超过 80%，则记录这个信息到表 rest_t。下面执行这个过程。

```
SQL> exec monit_res;
var_value: 80.5
var_value: 17.444
var_value: .915
var_value: .017
```

```
var_value: .056
var_value: .056

PL/SQL 过程已成功完成。
```

上面的输出记录了所有满足条件的资源使用率，而超过 80%使用率的资源则被记录到表 res_t 中，下面开始查询该表，如实例 17-67 所示。

实例 17-67　查询表 res_t。

```
SQL> select * from res_t;

MYNAME                     MYVALUE
---------------------------------------- ----------
processes                      80.5
```

这个记录告诉我们资源 processes 已经到了警告阈值，需要适当增大该参数的数值。至于如何修改这个参数，那就是数据库管理的内容了，这里不再介绍。

17.5　本章小结

本章介绍了 Oracle 的常用工具，其中包括 Oracle 提供的工具包和我们根据需要设计的工具包。对于 Oracle 提供的包主要包括调度管理包、审计包、解析执行计划包和 DBMS_HPROF 包。了解并掌握这些包的使用对于日常管理、调试程序以及优化十分必要。而对于 Oracle 没有提供的包，可以根据需求自己编写，这些包的基础都是动态性能视图，根据不同的需要着重分析了警告日志包、数据库维护包、历史数据包的作用和创建过程。